Polytechnic University of Catalonia
(UPC) - BarcelonaTech

Barcelona Schools of Informatics (FIB)

Extending *OmpSs-2* with flexible task-based array reductions

Ferran Pallarès

Master in Innovation and Research in Informatics

High Performance Computing specialization

supervised by

Vicenç Beltran (BSC-CNS) Sergi Mateo (BSC-CNS)

Eduard Ayguadé (DAC)

Barcelona Supercomputing Center - *Centro Nacional de Supercomputación* (BSC-CNS)

15th January, 2019

Acknowledgments

I would like to express my sincere gratitude to my tutor Eduard Ayguadé who trusted and gave me the opportunity to get involved in the Barcelona Supercomputing Center (BSC) and get to work in the field of High-Performance Computing (HPC).

I sincerely thank Vicenç Beltran, my director, for encouraging me to get engaged in this project, providing me with the necessary guidance to go through it.

The completion of this study could not have been possible without the expertise and invaluable help of my co-director Sergi Mateo and Josep Maria Pérez, who have been always willing to help and clear any of my doubts out. Thank you for having the patience of teaching me.

Finally, I need to thank with special affection my partner Laura, my family and friends. They have not only supported me this time, but have always been at my side, believing in me and in my decisions. I would not be where I am without them.

Abstract

With the evolution of multicore architectures, the urge for parallelising applications in the field of High-Performance Computing (HPC) has not stopped growing. Reductions, for their importance in complex scientific applications, have been studied for many years. However, when it comes to parallel programming models, reductions are still an ongoing subject of research.

After a brief introduction to different parallelisation strategies for task reductions, this thesis presents a flexible scheme for parallelising reductions of arrays in the context of *OmpSs-2*, a task-based programming model similar to *OpenMP*.

The contributions of this project include a formal specification of task reductions of arrays, in which the existent `reduction` clause is enhanced to support this feature. A new `weakreduction` clause is also introduced to support more complex scenarios.

A complete implementation of the proposed extension is developed using the *Nanos6* runtime library and the *Mercurium* source-to-source compiler. The key points of this implementation are presented, justifying any design decisions taken throughout the process. This implementation is then used as a baseline from which a wide range of optimisations has been explored.

Finally, a thoughtful evaluation using a set of relevant benchmarks shows how the provided implementation outperforms the state-of-the-art *OpenMP* parallelisation in most scenarios.

Contents

List of Figures

List of Code Listings

List of Tables

Acronyms

API Application Programming Interface. 6, 15

BSC Barcelona Supercomputing Center. i, 1, 5, 14, 57

DSP Digital Signal Processor. 1

FPGA Field-Programmable Gate Array. 1

GPU Graphics Processor Unit. 1, 24, 67

HPC High-Performance Computing. i, 1, 2, 67

ISA Instruction Set Architecture. 51

MPI Message Passing Interface. 1

NUMA Non-Uniform Memory Access. 17, 52, 54, 57, 61, 63

RaW Read-after-Write dependence. 8

SIMD Single Instruction Multiple Data. 51

SLURM Simple Linux Utility for Resource Management. 56

StarSs Star SuperScalar. 5

TPACF Two Point Angular Correlation Function. iv, v, vii, 60, 61, 72, 75–79

WaR Write-after-Read dependence. 8, 67

WaW Write-after-Write dependence. 8

1. Introduction

As of today, we are witnessing the very end of the *Moore's Law*[22] and it has already been about ten years since the breakdown of the power proportion known as *Dennard scaling*[11]. As a consequence of this, many chip manufacturers have switched their interest to multicore and many-core processors and, with it, the need for parallelising applications to use the available resources efficiently has rapidly grown. This has been especially the case for HPC applications and scientific research, where the need to develop highly optimised software is even greater.

Moreover, these multiprocessor systems are often enhanced with extra hardware resources used to accelerate some specific tasks or recurrent programming patterns. Such accelerators traditionally included Graphics Processor Units (GPUs), but nowadays are enriched with Field-Programmable Gate Arrays (FPGAs), Digital Signal Processors (DSPs), and even more specific accelerators designed for Computer Vision, Cryptography or Artificial Intelligence, to name some.

With such complex and heterogeneous systems, the programmers are often overwhelmed and, many times, the resources end up being underused. On the other hand, when a programmer optimises an application for one specific architecture, it becomes so specialised that it makes it painfully hard and time-consuming to be ported to another architecture, or even to an upgraded version of the same architecture.

Some programming models have emerged under the promise of providing competitive performance and a better resource usage without compromising portability, providing abstractions from the underlying hardware details. Some of these programming models, like MPI[12], focus on providing a paradigm for distributed memory between different compute nodes, while others, like *OpenMP*[3], *OmpSs*[6], and *OmpSs-2*[5], are centred around shared memory within the node.

OmpSs-2 is a shared-memory task-based programming model specification composed of a set of directives, library routines and environmental variables used to specify high-level parallelism in C/C++ and Fortran programs.[5] *OmpSs-2* is inspired by *OpenMP* and resembles to it in many points, even though *OmpSs-2* is completely designed as a task-based model. This model is being actively developed by the Programming Models group of the Computer Sciences department of the BSC.

In HPC applications we often find recurring patterns that are parallelised alike, using the same mechanisms provided by the chosen programming models. The reduction pattern is a well-known algorithmic pattern whose parallelisation is challenging. Although many efforts have been made to improve the execution of this pattern, some complex scenarios require further optimisations.

This project aims to propose a flexible scheme for computing parallel reductions in the *OmpSs-2* programming model, making special emphasis on reductions involving dense arrays. We contribute a formal specification of task array reductions as well as a complete implementation that we evaluate through a set of relevant benchmarks.

1.1 Document structure

This document serves the purpose of giving the reader a broad view on the work carried out throughout this project.

After a brief introduction to the reduction pattern and the *OmpSs-2* programming model, this document presents a contextualization section where different parallelisation strategies for task reductions are introduced.

The main contributions of this project are divided into three chapters. The first chapter provides a formal specification of the task array reductions in the context of the *OmpSs-2* programming model. This specification should suffice for its users to understand and use the feature unmistakeably, whereas any vendor that implements the model to develop their implementation complying to it.

The second chapter introduces a complete implementation that complies to the previous specification. The general design and the decisions that have lead to it are discussed in this chapter. The following chapter presents some optimisations that have been developed on top of the base implementation. While none of those optimisations impose any change on the way task reductions are used, they make them more relevant by providing mechanisms to speed-up certain recurrent situations.

Next, we evaluate the presented implementation and compare its performance on a set of relevant benchmarks in the field to the reference *Intel OpenMP* implementation using the *MareNostrum4* supercomputer.

Finally, the conclusions derived from the project are exposed, and further extensions along the work and the topic in general are suggested. In addition, references and appendixes are compiled in attached sections so that the reader can verify the information sources or delve into the project work if interested.

1.2 The reduction pattern

Reductions are a common algorithmic pattern found in many scientific applications[9]. In a reduction, a collection of objects are *reduced* to a single object by combining them pairwise with a binary operator.[23,18,19]

A reduction is a successive update of a variable, defined as:

$$var := op(var, expr)$$

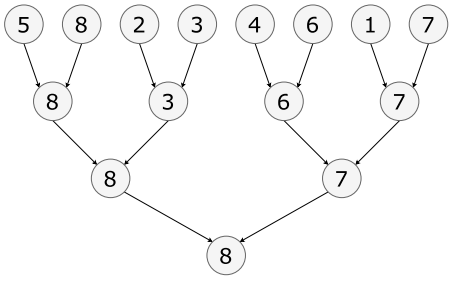
Where *var* is the variable where to combine the objects, *op* is the binary operator and *expr* is an expression that does not modify the value of *var*.[23]

While reductions do not enforce that the operator satisfies the associative and commutative properties by definition, most times it does. When assuming this, parallel implementations of the pattern are possible, and considerable speedup can be obtained accordingly.[23]

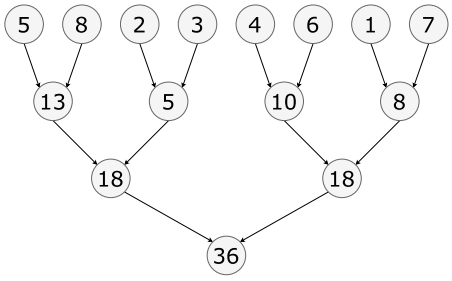
Special care must be taken when dealing with floating point arithmetic, as it is only approximately associative. In other words, different order in the operands can give different results due to round-off errors.[23,29]

Reductions are characterised for having non-atomic updates involving an accumulator variable and an expression, requiring exclusive access to ensure data consistency and making their execution computationally expensive and parallelization challenging.[23,9]

In figure 1.1 we can see a reduction pattern for *max* and *add* operators.[23]



(a) Reduction pattern for *max* operator



(b) Reduction pattern for *add* operator

Figure 1.1: Reduction pattern examples

1.2.1 Array reductions

Reductions of arrays, or more commonly referred to as array reductions, are nothing but an extension of the reduction pattern definition that affects not one but multiple subsequent elements in memory.

A simple example of a reduction of arrays is the computation of a histogram. For illustration purposes, imagine we are interested in knowing the frequency of each letter appearing in an English text. Code listing 1.1 shows an example C code for computing the letter frequency of a given text, while figure 1.2 shows the expected resulting histogram.[30] In the code snippet `letter_freq` is an array of counters for each letter in the English alphabet. The process of updating such an array when counting the individual letters in the text is an array reduction.

```
void letterCount(char *text, int length,
        float letter_freq[26])
{
    int total_letters = 0;

    for (int i = 0; i < length; ++i)
    {
        char letter = text[i];
        if (letter >= 'a' && letter <= 'z') {
            letter_freq[letter - 'a']++;
            total_letters++;
        }
        else if (letter >= 'A' && letter <= 'Z') {
            letter_freq[letter - 'A']++;
            total_letters++;
        }
    }

    for (int i = 0; i < 26; ++i)
        letter_freq[i] /= total_letters;
}
```

Listing 1.1: Letter counting

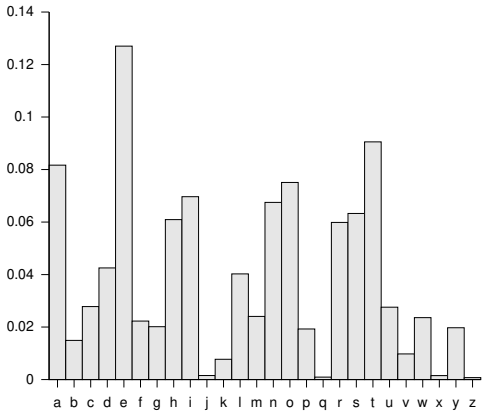


Figure 1.2: Relative frequencies of letters in an English text

1.3 The *OmpSs-2* programming model

OmpSs-2 is the second generation of the *OmpSs* programming model, a task-based programming model orientated at parallelising shared-memory applications in C/C++ and Fortran. Both programming models are developed at the *Programming Models* group in the BSC. The name originally comes from the union of two other programming models: *OpenMP* and *Star SuperScalar (StarSs)*. The design principles of these two programming models constitute the fundamental ideas used to conceive the *OmpSs* philosophy.[5,6]

On the one hand, *OmpSs* takes from *OpenMP* its viewpoint of producing a parallel version from an initial sequential program by introducing annotations in the source code. These annotations do not explicitly affect the semantics of the program, instead, provide the necessary information so that the compiler can produce a parallel version of the program. This functionality allows the users to perform an incremental parallelisation of their applications by adding new directives to specify the parallelism on different parts of the application.[5]

On the other hand, *OmpSs* is inspired by *StarSs* in its thread-pool execution model, which differs from the *OpenMP* fork-join parallelism. Moreover, *StarSs*, as well as *OmpSs*, targets heterogeneous architectures through launching native kernels to the accelerators, while *OpenMP* gives its support through compiler code generation (i.e. the compiler is responsible for generating the native kernels).[5]

Both *StarSs* and *OmpSs* offer asynchronous parallelism in the form of tasks as the primary mechanism of expressing parallelism. This mechanism is enhanced with synchronisation capabilities by means of task dependences, which allow expressing the task execution order and enabling the look-ahead instantiation. Oppositely, *OpenMP* was designed to provide synchronous parallelism via its fork-join model, and it started to adopt such features at its version 4.0.[5]

Finally, *StarSs* and *OmpSs* try to be more implicit that *OpenMP*. In *OpenMP*, the developer is responsible for explicitly defining which regions are going to be executed in parallel and to define the synchronisation points for the threads executing those regions. In contrast, in the environment provided by *StarSs* (which is inherited in *OmpSs*) the parallelism is implicitly created from the beginning of the execution, and the developer is only responsible for specifying the meaningful basic blocks in the program as tasks and the data they access. Given this information, the dependences between tasks can be figured out and thus the parallelism is computed transparently from the developer guaranteeing a proper execution. This mechanism favours not only an easier, higher-level parallelisation by the developer, but also a much richer expression of parallelism that makes applications exploit the parallel resources more efficiently.

The reason-to-be of the *OmpSs-2* programming model and probably the most ambitious of its objectives is to extend the *OpenMP* programming model with new directives, clauses and Application Programming Interface (API) services or general features to better support asynchronous data-flow parallelism and heterogeneity.[5]

Many ideas initially conceived in *OmpSs* have been already introduced in the *OpenMP* standard. Figure 1.3 summarizes them.

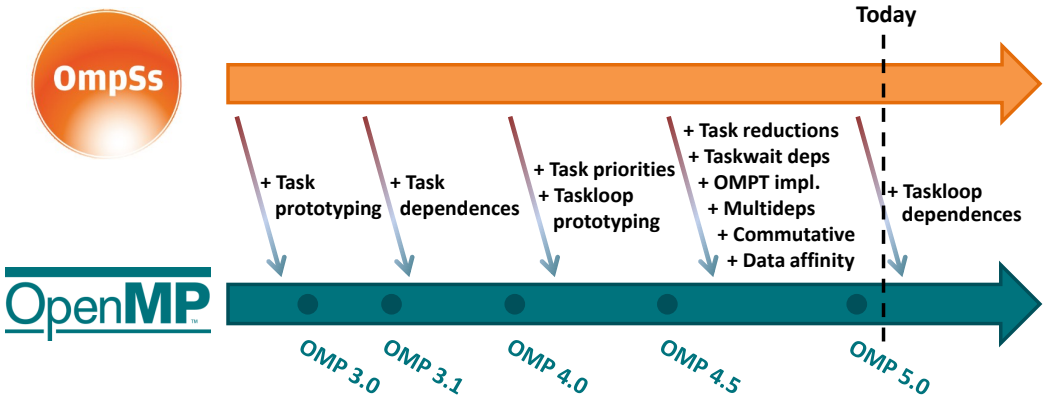


Figure 1.3: *OmpSs* contributions to *OpenMP*

Some of the contributions of the *OmpSs* programming model that are already part of the *OpenMP* standard include:

- Task dependences (included in *OpenMP* 4.0)
- *OpenMP* SIMD extensions (included in *OpenMP* 4.0)
- Task priorities (included in *OpenMP* 4.5)
- Task reductions (included in *OpenMP* 5.0)
- `mutexinoutset` dependence type (called `commutative` in *OmpSs*) (included in *OpenMP* 5.0)

1.3.1 Basic concepts in *OmpSs-2*

The most important concept in *OmpSs-2* is most probably the *task*: A task is a specific instance of executable code and its data environment that can be executed immediately or scheduled for a delayed execution.

Tasks are the elementary unit of work, the minimum execution entity that can be managed independently by the runtime scheduler (although a single task may be blocked and resumed at specific scheduling points). In *OmpSs-2*, tasks are created using the appropriate compiler directive, which is known as the `task` construct. The syntax of the `task` construct for C/C++ is the following:

```
#pragma oss task [clause[ [,] clause] ...]
structured block
```

In the general case, the structured block that follows the `task` construct is to be executed asynchronously. Whenever a thread encounters a task, it instantiates it and resumes its execution after the construct. The task instance can be executed either by that same thread at some other time or by another thread. The semantics can be altered through additional clauses and through the properties of the enclosing environment in which the task is instantiated.[5]

The valid clauses for the `task` construct are:

- `private(`<*list*>`)`
- `firstprivate(`<*list*>`)`
- `shared(`<*list*>`)`
- `depend(`<*type*>*:* <*memory-reference-list*>`)` | <*depend-type*>`(`<*memory-reference-list*>`)`
- `reduction(`<*reduction-identifier*>*:* <*item-list*>`)`
- `priority(`<*expresion*>`)`
- `cost(`<*expresion*>`)`
- `if(`<*scalar-expression*>`)`
- `final(`<*scalar-expresion*>`)`
- `label(`<*string*>`)`
- `wait`

As in *OpenMP*, explicit task synchronisation points can be achieved in *OmpSs* by means of the `taskwait` directive:

```
#pragma oss taskwait [clause[ [,] clause] ...]
```

The `taskwait` directive allows waiting for all previously created tasks at the moment the directive is encountered and continuing from that point once the tasks have finished. In order to wait for a limited subset of tasks, the directive allows specifying dependences using the dependence clauses, as it is done for the `task` directive. If fine-grained synchronisation between tasks is desired, then the mechanism of dependences should be used instead.[5]

1.3.2 The *OmpSs-2* dependence model

Dependences are the *OmpSs-2* mechanism that allows determining the data flow and parallelism of a program.

When an *OmpSs-2* program is being executed, the underlying runtime environment uses the data dependence information and the creation order of each task to perform dependence analysis. This analysis produces execution-order constraints between the different tasks which results in a correct order of execution for the application. We call these constraints task dependences. In other words, dependence is the relationship existing between predecessor tasks (must execute before) and one of its successor tasks (must execute after).[5]

Each time a new task is created its dependences are matched against of those of existing tasks. If a dependence, either Read-after-Write dependence (RaW), Write-after-Write dependence (WaW) or Write-after-Read dependence (WaR) is found, the task becomes a successor of the corresponding tasks. This process creates a task dependence graph at runtime. Tasks are scheduled for execution as soon as all their predecessor in the graph has finished (which does not mean they are executed immediately) or at creation if they have no predecessors.[5]

Data dependences are defined in the `task` construct using either the `depend` clause (as it is done in *OpenMP*) or by the *OmpSs* short form using the dependence name directly. The clauses allow specifying, for each task, what are the intentions of the task regarding that data. For instance, they may declare the intention for a task to read some data or the intention to produce some other data. Such intentions are processed by the runtime to materialise the actual dependences between tasks. Whether the task really uses that data in a specified way its the programmer responsibility.

The usual scope of the dependence calculation is restricted to that determined by the enclosing (possibly implicit) task. That is, the contents of the `depend` clause of two tasks can determine dependences between them only if they share the same

parent task (referred to as sibling tasks). In this sense, tasks define an inner and independent dependence domain into which to calculate the dependences between its direct children.[5]

The `depend` clause admit the following keywords: `in`, `out`, `inout`, `concurrent`, `commutative` and `reduction`. The keyword is followed by a colon and a comma separated list of elements (memory references).[5] The syntax permitted to specify memory references for C/C++ using the `depend` clause is described as follows:

```
depend(<type>: <memory-reference-list>)
```

while the short *OmpSs* form syntax is:

```
<type>(<memory-reference-list>)
```

In both, *memory-reference-list* refers to an *lvalue*[24,10] expression using the syntax of the underlying programming language, while the dependence *type* can be either one of the following basic types:

- `in`: An `in` dependence type enforces a dependence over a previously created sibling task which defines either an `out` or `inout` dependence type over the same memory reference.[5]
- `out`: An `out` dependence type enforces a dependence over a previously created sibling task which defines either an `in`, `out`, `inout`, dependence type over the same memory reference.[5]
- `inout`: An `inout` dependence type combines the semantics of the `in` and `out` dependence types, enforcing all dependences described in the previous two points.[5]

Table 1.1 displays the detailed interaction between the described dependence types.

		predecessor		
		in	*out*	*inout*
successor	*in*		RaW	RaW
	out	WaR	WaW	WaR+WaW
	inout	WaR	RaW+WaW	RaW+WaW+WaR

Table 1.1: Interaction between dependence types

In addition to the described basic types, the dependence clauses also allow the following special types. These types are specialisations of the `inout` type with some relaxations that allow a greater degree of parallelism:

- `concurrent`: The `concurrent` dependence type behaves as the `inout` type with respect to `in`, `out` and `inout` types, but has the particularity that no dependences are enforced over other sibling tasks that define a `concurrent` type over the same memory reference.[5]
- `commutative`: The `commutative` dependence type behaves as the `inout` type with respect to `in`, `out` and `inout` types. It also enforces a dependence over other sibling tasks that define a `commutative` type over the same memory reference, but this dependence allows any order of execution between those tasks (as opposed to creation order). Any permutation ordering of those tasks annotated with `commutative` is correct, as long as only one of those tasks is executed at a time.
- `reduction`: As far as the interaction between dependence types is concerned, the `reduction` type behaves just as the `concurrent` type. The difference between them is that a task annotated with a `reduction` clause will also be responsible for computing a reduction, but this has no implications from the point of view of the dependence model.

In the code listing 1.2 we can see an example of a taskified code in *OmpSs-2* where the data accesses have been annotated. The corresponding generated dependence graph can be seen in figure 1.4. For our example, tasks `T2` and `T3` could run in parallel provided enough resources are available.[23]

1.3.2.1 Task nesting and dependences

OmpSs-2 extends the tasking model of *OmpSs/OpenMP* to support fine-grained dependences across different nesting levels, which enables the effective parallelization of applications using a top-down methodology.[5]

Being able to combine task nesting and dependences is important for programmability. Outer tasks should contain a combination of elements to protect their own accesses and elements that are only needed by their subtasks. This is necessary to avoid data-races between subtasks with different parents.[5]

The inclusion in the outer task of dependence clauses for elements required only by subtasks effectively link the dependence domain of the task with that of its subtasks, which otherwise would be disconnected.[5] Code listing 1.3 shows an *OmpSs-2* program with nested tasks and dependences. Note that, as opposed to *OpenMP*, the `taskwait` directive will deeply wait for all previously created tasks, including all their descendants.

```
int a, b = 0, tmp;

#pragma oss task inout(a, b) label(T1)
{
    b += compute_value(...);
    a += compute_value(...);
}

#pragma oss task in(a) out(tmp) label(T2)
tmp = a + compute_value(...);

#pragma oss task in(a) label(T3)
print(a);

#pragma oss task in(b, tmp) label(T4)
print(b*tmp);

#pragma oss task inout(a, b) label(T5)
{
    b *= compute_value(...);
    a *= compute_value(...);
}
#pragma oss taskwait
```

Listing 1.2: Data flow example in *OmpSs-2*

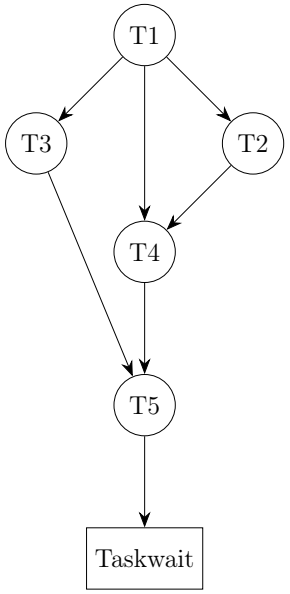


Figure 1.4: Task dependences in *OmpSs-2*

Once a task has finished its execution, the runtime is aware that it will not create further subtasks neither require the enforcement of the dependences for itself any longer. In a scenario with task nesting and dependences, this knowledge allows the runtime to preserve only the dependences needed by its live subtasks, providing a fine-grained release of the dependences that could allow subsequent tasks to begin their execution sooner.[5]

Even with the fine-grained release optimisation, the elements of the dependence clauses that are not needed for the outer task itself delay its execution, and hence the instantiation of its subtasks. In order to cope with this problem, the *weak-dependences* were introduced. `weak` is a modifier that can be prefixed to the dependence types defined above to indicate that the enforcement of the dependence is not directly required by the task, but for some nested subtask instead. The outer task is then allowed to be executed even if its weak-dependences are not yet satisfied, while still linking the outer domain of dependences to the inner one. The dependences will eventually be enforced for the inner subtasks annotated with the regular (strong) dependence types.[5]

Code listing 1.4 improves the parallelization shown in 1.3 by using `weak` dependences.

```
int a = 0, b, c;

#pragma oss task inout(a), out(b)
{
    #pragma oss task inout(a)
    a++;

    #pragma oss task out(b)
    b = foo();
}

#pragma oss task in(a, b) out(c)
{
    c = b;

    #pragma oss task in(a)
    printf("A: %d\n", a);
}
#pragma oss taskwait
```

Listing 1.3: Nested dependences in *OmpSs-2*

```
int a = 0, b, c;

#pragma oss task weakinout(a) out(b)
{
    #pragma oss task inout(a)
    a++;

    #pragma oss task out(b)
    b = foo();
}

#pragma oss task weakin(a) in(b) out(c)
{
    c = b;

    #pragma oss task in(a)
    printf("A: %d\n", a);
}
#pragma oss taskwait
```

Listing 1.4: Weak-dependences in *OmpSs-2*

`weak` dependences, combined with the fine-grained release of dependences, merge the inner dependence of a task into that of its parent. Since this happens at every nesting level, the result is equivalent to an execution in which all tasks had been created in a single dependence domain.[5]

1.3.2.2 Region dependences

As stated in section 1.3.2, memory references annotated in the dependence clauses evaluate to *lvalues*[24,10]. This is important because it implies that the expressions allowed in such clauses do not necessarily refer to a single position in memory, but rather a range of elements. Then, the runtime is responsible for computing the exact intersection between the memory references annotated in the different tasks in order to figure out the real task dependences. This is the reason why precisely annotating the dependences is a fundamental aspect of the *OmpSs-2* programming model.[5]

In addition, *OmpSs-2* dependence clauses allow extended expressions from those *lvalues* of C/C++ with the objective of increasing the programmability and promoting cleaner code[5]. Two different extensions are allowed:

- **Array sections** allow to refer to multiple elements of an array (or pointed data) in single expression. There are two forms of array sections:
  - `a[lower : upper]`: In this case all elements of `a` in the range of `lower` to `upper` (both included) are referenced. If no `lower` is specified it is assumed to be `0`. If the array section is applied to an array and `upper` is omitted then it is assumed to be the last element of that dimension of the array.
  - `a[lower; size]`: In this case all elements of `a` in the range of `lower` to `lower + (size - 1)` (both included) are referenced.
- **Shaping expressions** allow to recast pointers into array types to recover the size of dimensions that could have been lost across function calls. A shaping expression is one or more `[size]` expressions before a pointer.

Code listing 1.5 displays both extensions in use.

```
void sort(int n, int *a) {
    if (n < small)
        seq_sort(n, a);

    #pragma oss task inout(a[0:(n/2)-1]) // Array section, eq.  to inout(a[0;n/2])
    sort(n/2, a);

    #pragma oss task inout(a[n/2:n-1]) // Array section, eq.  to inout(a[n/2;n/2])
    sort(n/2, &a[n/2]);

    #pragma oss task inout([n]a) // Shaping expression
    merge(n/2, a, &a[0], &a[n/2]);
}
```

Listing 1.5: Array sections and shaping expressions in use

Note that these extensions are only for C/C++, since Fortran already supports array sections natively.

1.3.3 Data sharing attributes in *OmpSs-2*

As in *OpenMP*, *OmpSs* and *OmpSs-2* allow specifying the explicit data sharing attributes for the variables referenced in a construct using the following clauses:

- `private(`<*list*>`)`
- `firstprivate(`<*list*>`)`
- `shared(`<*list*>`)`

The `private` and `firstprivate` clauses declare one or more variables to be `private` to the construct (i.e. a new variable will be created). All internal references to the original variable are replaced by references to this new variable. Variables privatised using the `private` clause are uninitialized when the execution of the construct begins. Variables privatised using the `firstprivate` clause are initialised with the value of the corresponding original variable when the construct was encountered.[5]

The `shared` clause declare one or more variables to be `shared` to the construct (i.e. the construct still will refer the original variable). Programmers must ensure that `shared` variables do not reach the end of their lifetime before other constructs referencing them have finished.[5]

When the variable is not referenced by any of the explicit data sharing clauses it is considered to have an implicit data sharing attribute. In particular, if the variable appears in a dependence clause, the variable will be `shared`.[5]

1.3.4 Reference implementation

The reference implementation of the *OmpSs-2* programming model is based on the *Mercurium* source-to-source compiler and the *Nanos6* Runtime Library:

1.3.4.1 *Mercurium* compiler

The *Mercurium* compiler is a *source-to-source* compiler developed in the Programming Models group of the Computer Sciences department of the *BSC*. *Mercurium* currently supports C, C++ and Fortran languages.[4]

Mercurium provides the necessary support for transforming the high-level directives into a parallelized version of the application. In detail, *Mercurium* can be used to transform the *OmpSs-2* annotations included in the input source code into standard C, C++ and Fortran: It generates the required routines and calls to the *Nanos6* runtime library. Once the *OmpSs-2* annotations have been *lowered* it handles the control to the native compiler, so it generates the final executable binary.

While its main purpose is to enable the use of *OpenMP*, *OmpSs* and *OmpSs-2* by interpreting the corresponding programming model `pragmas`, *Mercurium* is also capable of detecting and applying some compile-time optimisations that often result in more efficient code.

1.3.4.2 *Nanos6* runtime library

The *Nanos6* runtime library, often referred to as just *runtime*, is the reference library that provides runtime support for the *OmpSs-2* programming model. It is the software piece responsible for providing the services to manage all the parallelism in the user-application, while implementing the API calls defined in the programming model standard.

Nanos6 provides the fundamental services that make possible the execution of tasks. Those include the creation and instantiation of tasks, the registration of their dependences and the computation of the task dependence graph, which is then used for scheduling the tasks on the resources, honouring the correct execution order while implementing different scheduling policies. It also provides the components that manage the system resources and provide support for accelerators and heterogeneity.

A range of operation modes and configuration environmental variables exist for the runtime to be able to tune its behaviour during the program execution, enable debugging facilities or instrument the execution for later in-depth analysis[21]. As far as the latter is concerned, *Nanos6* provides a dedicated *Extrae*[14] instrumentation mode capable of generating *Extrae* events registering both the runtime activity and the available performance hardware counters at meaningful points during the execution. Those traces can be then analysed using *Paraver* to better understand the behaviour of the *Nanos6* runtime and the executed application under whichever specific circumstances and carry out performance evaluation.

2. Contextualization

In this section, we aim to offer the reader some background on the different techniques that can be used to implement reductions in the context of a task-based programming model like *OpenMP* or *OmpSs*. First, we focus on strategies that can be used to provide a base implementation, and then we present two different approaches that extend it to support nested reductions.

2.1 Parallelisation techniques for task reductions

In this section we will briefly describe the reduction parallelisation strategies that are most relevant for this work, as well as discussing their weak and strong points and finally giving an intuition of what kind of applications they best suit.

2.1.1 Parallelising task reductions through *synchronisation*

In a *synchronisation*-based task reduction parallelization strategy, no auxiliary storages are used to speed-up the computation of the reduction. Instead, all tasks will be accessing the original data directly, and thus their execution will require synchronisation in order to avoid memory consistency problems.

The simplest mechanism used to ensure a correct computation of the reduction pattern consists in serialising the tasks that participate in the reduction, avoiding any data-race. Even though this strategy does not provide any extra parallelism and may seem irrelevant at first, if the program already provides enough parallelism to completely utilise the available resources this strategy will probably be the most effective, as it does not add any overhead.

A different approach that lacks auxiliary storages, but does allow the reduction to be computed in parallel, can be implemented by using explicit synchronisation mechanisms. For example, *locks* can be used to define mutual exclusion sections in the code that will ensure all updates to the reduction variable are done exclusively. In contrast,

such mechanism can become a source of contention if many reduction tasks are executed in parallel. In order to reduce this effect, atomic instructions can be used to consistently update the reduction data, granted that they are available in the underlying architecture. Although atomic instructions are much more efficient than *locks*, their scope of application is limited to the supported instructions and the code visibility. In another words, the required reduction operation may not be implementable using atomic instructions, or it may be found in a library function that is called from within the task.

Synchronisation may be sped-up by relaxing the memory consistency model. By fulfilling the associative and commutative properties, reduction operations are suitable to use those models and, by doing so, benefit from the performance gains they offer, especially in architectures implementing Non-Uniform Memory Access (NUMA).

The synchronisation-based parallelisation mechanism suits applications performing sparse reductions of huge arrays, where the privatisation cost would be forbidding (up to the point where system memory is exhausted) and the probability of multiple tasks accessing at the same memory position at the same time is low.

The parallelisation of task reductions through synchronisation can be fully implemented as a compiler transformation.

2.1.2 Parallelizing task reductions through *privatisation*

In a *privatisation*-based task reductions parallelisation strategy, auxiliary storages are used to speed-up the computation of the reduction. Those auxiliary storages will be used to break the data dependences and allow multiple tasks to be executed in parallel without having to deal with synchronisation nor memory consistency problems. As far as the data dependences are concerned, privatisation is similar to the classical *renaming* technique.

Contrary to *renaming*, reduction privatisation does not require its storages are a copy of the original data, but to be initially set to the neutral value of the reduction instead. This process is known as *initialization*.

Tasks will contribute to the reduction by making updates to the private storage they have been assigned (only associative and commutative operations). Once their execution is over, the contribution of those tasks will be accounted for by merging all private storages together, using the reduction operator in a process known as *combination*. Again, the properties of the reduction pattern will guarantee a correct result regardless of the order in which the private storages are combined.

In a full privatisation strategy, it is required that each task executed in parallel at a given point will use a different private storage. On the other hand, there are other strategies in which a group of tasks share a single private storage while another group

share a different private storage, requiring synchronisation only within the group of tasks. Strategies like the previous sacrifice parallelism in favour of a reduced memory impact.

Considering the number of private storages and the criteria followed to assign them upon task execution, the following sections aim to introduce the most commonly-used privatisation strategies.

2.1.2.1 Task-privatisation strategy

In a task-privatisation strategy, as many private storages as tasks are allocated. A new private storage is used for the execution of each task, and so it needs to be initialised to the neutral element. Each storage will hold the contribution of a single reduction task which, at the end of its execution, will be combined into the original data. This strategy reduces the number of updates to the original data to one per task, which is usually synchronised using atomic instructions.

This strategy suits reductions of scalar variables or small arrays or structures, as the amount of memory to privatise is small and the cost of allocating, initialising and combining the private storage is almost negligible.[23] For larger arrays, this cost will increase proportionally to the array size, and will end up weighing more than the task execution itself. In addition, if the programming model allows reduction tasks to be yielded and rescheduled, their storages cannot be freed, potentially causing memory hogging problems. When large array reductions are to be computed, the previous two points usually tip the balance over other strategies with a smaller memory footprint.

Similarly to synchronisation-based reduction parallelisation, task-privatisation can be fully implemented as a compiler transformation.

2.1.2.2 CPU-privatisation strategy

A CPU-based privatisation strategy consists in allocating as many private storages as number of CPUs available to the application. When the reduction is registered in the system, a private storage is allocated for each CPU. Then, when a task of the registered reduction is executed for the first time in a CPU, the CPU private storage is initialised to the neutral element so, for a given reduction, the initialisation cost is only paid once per CPU. This storage is later reused for other tasks participating in the same reduction until all reduction tasks have been executed and the combination is triggered from within the runtime.

For instance, the combination can be triggered when a successor task that has a dependence over the data is to be executed, or a `taskwait` directive is found. This

privatisation strategy is completely orchestrated by the runtime, that handles the allocation, initialisation and combination of the private storage. For this reason, and contrary to task-privatisation, the combination process can be controlled, and no external synchronisation mechanisms are required.

CPU-privatisation should perform well on dense reductions of medium-to-big arrays. Scalar reductions will have to pay the overhead of the runtime calls, which will be inevitably greater that a fully-compiler transformation. On the other hand, for reasonably-sized arrays, the overhead of the privatisation should be much smaller than in the task-privatisation strategy, since it is now proportional to the number of CPUs, and those are expected to be much fewer than the number of tasks.

However, cutting the number of private storages down to the number available CPUs will not suffice for all scenarios: to privatise huge arrays will possibly require more memory than it is available in the system, even if the number of required storages is low. Moreover, if the reduction is sparse, the cost of allocating, initialising and combining many unaccessed elements in the privatised storage will not compensate for being able to compute a few other faster. Besides, having the private storages bound to CPUs implies that the task must be executed entirely on the CPU it was first scheduled, or otherwise ensure that the reduction variable points to the proper CPU private storage after each task yield point.

This privatisation strategy requires compiler support to add function calls to the runtime and make the reduction variable point to the privatised storage, but the mechanism logic is essentially controlled by the runtime.

2.1.2.3 Hybrid CPU-and-task privatisation strategy

An hybrid CPU-and-task privatisation is possible and corresponds to a combination of both previous strategies. In this combined strategy, the task accumulates its contribution to a task-private storage that, instead of being combined directly onto the original data, it is combined into a CPU-private storage managed by the runtime.

By doing so, this strategy avoids dealing with reduction tasks that migrate to other CPUs at yield points, since the task-private storage will still be valid when the task is resumed. The CPU-private storage will only be accessed once, when the task has finished the execution and combines the task-private storage to it.

By performing task-privatisation, this strategy is subjected to the limitations explained in section 2.1.2.1 and thus it is also restricted to scalar reductions and small arrays.

2.2 Techniques for enabling nested array reductions

The aim of this section is to describe the implications of specifying reductions over nested tasks. In the following subsections, we discuss the different nesting paradigms, as well as the implications that arise from combining them with the previous parallelisation techniques. A general explanation of task nesting and a description of how dependences behave in nested tasks in *OmpSs-2* are provided in section 1.3.2.1.

2.2.1 Local reduction nesting paradigm

The local reduction nesting paradigm is only possible in reduction parallelisation strategies that perform privatisation over the reduction data. The paradigm consists in registering nested reductions in terms of the parent reduction private storage; that is, the *original data* with respect to the nested reduction corresponds to some private storage from the parent reduction task.

While the computation of the subtask reduction should be computed independently from the parent reduction, the result of the inner reduction must be contributed to the parent reduction private storage at some point. Given that the parent task may be using its private storage for its own update operations, the subtask cannot perform this combination until both tasks have finished executing. Therefore, for this combination to work, no private storage used in any nesting level can be combined into its *original data* until all its subtasks have performed their combination. The described schema is known as *bottom-up combination*.

2.2.2 Global reduction nesting paradigm

The global reduction nesting paradigm consists in detecting when nested reductions are registered and being able to make nested tasks participate in the parent reduction, constituting a single, global reduction across multiple nesting levels.

All synchronisation-based strategies discussed in section 2.1 can support global reductions efficiently and without any extra consideration, making it the preferred paradigm for them. On the other hand, supporting this paradigm in privatisation-based strategies is more complicated (and probably inefficient too).

Detecting the situation alone requires an additional mechanism to translate from private storage addresses to the original data. This requirement is not trivial to fulfil in a compiler-driven task-privatisation strategy, providing that the privatisation may have been done in another compilation unit. On the other hand, tracking this information in the runtime for a CPU-privatisation strategy is an inefficient process, especially if we consider that this information needs to be kept for each private storage within the reduction and that many reductions can be executed simultaneously in the system.

3. Related work

This section intends to review the current literature on the topic in question: array reductions in parallel programming models. Previous studies and similar approaches to the problem are discussed, explaining similarities and differences.

This chapter is divided into two parts. First, for its relevance in this work, the reduction support in *OpenMP* is discussed in detail. Then, other works that present a different approach to the problem are commented.

3.1 *OpenMP* reductions

The *OpenMP* specification currently includes several mechanisms to perform array reductions. On the one hand, the `reduction` clause can be found in a `parallel` construct or in the `for` work-sharing constructs.[2] On the other hand, *OpenMP* 5.0 introduced reduction support for tasks and the *taskloop* construct.

For task reductions, the clauses `task_reduction` and `in_reduction` were added to the model. When used in combination with the `taskgroup` directive, the former behaves as a reduction scoping clause, and its purpose is to delimit the domain of a reduction. The latter can be used in the `task` construct and is defined as a reduction participating clause. Reduction participating clauses are used within the reduction domain to annotate a task participating in that reduction.[2]

Finally, the `taskloop` construct was extended accept both the `reduction` and `in_reduction` clauses. When used with a taskloop, the `reduction` clause acts as a scoping clause and a participating clause, meaning that the generated tasks will participate in a reduction whose domain is bounded by the taskloop. In contrast, when the `in_reduction` clause is used, the generated tasks will participate in a reduction previously defined by a reduction scoping clause.[2]

In the code listing 3.1 we can see how the dot product could be implemented using *OpenMP* task reductions.

```
float dot_product(const float *A, const float *B, unsigned int length)
{
    float result = 0;

    #pragma omp taskgroup task_reduction(+: result)
    for (unsigned int i = 0; i < length; i++)
    {
        #pragma omp task in_reduction(+: result)
        result += A[i]*B[i];
    }

    #pragma omp taskwait
    return result;
}
```

Listing 3.1: *OpenMP* reduction example with tasks

Task reductions in *OpenMP* were the main inspiration of this work. However, there are some limitations in *OpenMP* task reductions we would like to amend. First, in *OpenMP* the scope of a task reduction in very rigid: It is limited by a reduction scoping construct that dictates when the reduction begins and when the result must be ready. While this simplifies the implementation, we believe it does not blend well with the data-flow model proposed by *OmpSs-2*. In *OmpSs-2*, dependences can replace fixed synchronisation points providing a much more dynamic and fine-grained synchronisation between tasks. In a similar fashion, we believe that reductions should be much more flexible to integrate with the dependences component.

For instance, the array reduction mechanism presented in this work allows to compute the reduction in parts, as the data is needed by successor tasks. This is currently not possible in *OpenMP*.

In *OpenMP*, it is required for a variable specified in the `in_reduction` clause to be also specified in the `task_reduction` scoping clause. As a consequence of this, the memory references that make up a reduction are fixed and need to be known beforehand, at the reduction scoping clause. In our design, the reductions are not limited to a set of fixed memory references, but can rather be dynamically extended as tasks are created. For instance, code listing 3.2 shows a code where the data to be accessed in the reduction is computed dynamically.

To sum up, our work aims to relax some of the rigid constraints imposed by *OpenMP* in the task reductions mechanism in order to demonstrate that a more flexible model is possible. This model should resemble as much as possible to the dependence mechanism.

```
int i = 0;

#pragma omp taskgroup task_reduction(+: data[?!?]) // Unknown!
while (!finished) {
    #pragma omp task in_reduction(+: data[i:i+n])
    {
        [...]
    }

    finished = exec.hasFinished();
    i++;
}
```

Listing 3.2: *OpenMP* static memory references example

Besides, since the release of *OpenMP* 5.0 standard, *LLVM* is the only major vendor that publicly supports some of its new features (*Intel* and *GCC* still do not). In this sense, the implementation of task reductions we provide in this work may be used as a reference implementation for future comparisons.

3.2 Other works

[17] proposes region-based parallelisation techniques for irregular reductions on multicore architectures. They claim to simplify memory management for programmers by developing abstractions targeted to irregular reductions. They propose a compiler directive as an extension to *OpenMP* directives that can be used to annotate irregular reduction loops. The scope of their work is limited to detecting reductions in `for` loops for a single class of irregular reductions. On the contrary, our work is focused on supporting dense reductions of arrays in tasks.

[13] develops a compiler based method for the automatic detection of reduction operations. Their approach is based on a constraint formulation and solver they have implemented as a LLVM pass. Once discovered, they can automatically generate parallel code to exploit the reduction. This work is mostly focused on the automatic detection of reductions. However, this can only be done in specific applications where the compiler is able to see the whole code, and the reduction spans only a single compilation unit. While our approach requires compiler support too, we follow the *OpenMP* philosophy in which it is the user's responsibility to mark which are the reductions to be parallelised.

In [25], they present language constructs that allow programmers to express arbitrary reductions on user-defined data types. In their work, they are able to optimise the execution of reductions on GPUs. They are also able to perform optimisations in the presence of nested loops carrying reductions.

[23] presents an extension of the *OmpSs-2* tasking programming model in order to support task reductions for scalar types. First, the design and implementation of the concurrent, clause is presented, providing a user-driven mechanism to compute reductions. Then, the mechanism is extended into first implementation of the reduction clause based on a task privatisation strategy. Next, a different approach for the clause based on task-and-CPU privatisation is developed. Finally, both strategies are evaluated for a set of different architectures by subjecting them to different benchmarks for scalar reductions. This work has been used as a starting point for the contributions presented in this project.

4. Programming model extension

In this section we describe how the *OmpSs-2* programming model is extended in order to support array reductions. This section focuses on a formal specification of what is to be supported by the model, detailing what assumptions can be made by the user and defining the expected behaviour of the provided clauses, making special attention in their syntax and semantics.

By definition, a specification is a contract between the programming model user and the vendors that implement it. It specifies the interface to interact with the runtime library and provides a correctness framework for the user to follow. This specification intends to be implementation agnostic, and thus, any program written following this specification rules and statements is expected to work with any complying implementation of the *OmpSs-2* programming model.

4.1 Enhancing reduction clause expressibility

The `reduction` clause was first introduced in the *OmpSs-2* programming model to exclusively support scalar-type reductions in tasks.[23] During the course of this project, the `reduction` clause has been enhanced to support array reductions. This section explains these enhancements.

The syntax of the `reduction` clause is the following:

`reduction(`*<reduction-identifier>*`: `*<item-list>*`)`

Where the *item* was originally required to be a variable of scalar type. Naturally, the first necessary change was to relax this limitation and allow variables with an array or pointer type.

As far as the supported reduction operators are concerned, no changes have been made with respect to the original clause. The comprehensive list of supported operators can be found in appendix A.

Another enhancement applied to the `reduction` clause was to allow the use of `shaping expressions`. As explained in section 1.3.2.2, `shaping expressions` improve the clause expressibility by allowing to recast pointers into array types of any arbitrary number of dimensions. As opposed to `array sections`, when a `shaping expression` is used there is a guarantee that the referenced data will be continuous in memory, and that the base address of the specified *item* is still the address of the variable. Fulfilling these properties is necessary to be able to provide an efficient implementation of the array reduction mechanism, so it was decided that the `array sections` would not be supported for the first version of this specification. In the following sections of this chapter, the relevance of fulfilling these properties will be clarified.

4.2 Limitations derived from the privatisation

In *OmpSs-2*, reductions were designed to be generic. For this, use cases as the one shown in the code listing 4.1, where we find a call to an external routine within the task reduction, are to be supported. In the general case, the compiler will not be able to see where the actual reduction happens (it may be another *compilation unit* or even an external library), so it is not able to perform transformations on that part of the code.

```
double RRS[nbins];
double DRS[nbins];
[...]
for (rf = 0; rf < args.random_count; rf++) {
    #pragma oss task RRS[0:nbins], DRS[0:nbins])
    {
        // compute RR
        doCompute(random, npr, NULL, 0, 1, RRS, nbins, binb);

        // compute DR
        doCompute(data, npd, random, npr, 0, DRS, nbins, binb);
    }
}
```

Listing 4.1: Reduction task with a library call

The principal implication of this is that, in order to have a parallel implementation of task reductions, some privatisation strategy is required. For instance, a reduction mechanism that intends to replace all memory accesses within the task to use atomic instructions would not be able to support the previous example. On the other hand, a lock-based implementation would not be feasible either, because placing *locks* around the function calls could end up in a complete serialisation of the reduction tasks. A

much more reliable mechanism to use atomic instructions or locking would be a user-driven implementation using the `concurrent` clause. In this project, however, we are interested in a mechanism that is transparent to the user.

This specification does not have any further assumptions on how the privatisation should be performed, nor the number of copies or where will those be located. For instance, a task-privatisation strategy would be a simple privatisation strategy capable of fulfilling the specification, regardless the performance penalty that would probably incur by having to allocate, initialise and combine a possibly big array in every task participating in the reduction. It is the vendor's responsibility to provide an efficient privatisation mechanism.

As a consequence of the privatisation, any accesses outside the memory of the variable specified in the `reduction` clause (reduction variable) that is performed within the task using the declared base symbol lead to undefined behaviour (code 4.2). Furthermore, a reading operation over the memory of a reduction variable intending to read partial results will provide a different outcome depending on the implementation privatisation strategy, and thus is again undefined behaviour (code 4.3). Similarly, operating on the reduction variable with any operation other than the reduction operation specified in the `reduction` clause will also lead to undefined behaviour (code 4.4).

```
int x[10];

#pragma oss task reduction(+: [5]x)
{
    for (int i = 0; i < 10; ++i)
    {
        x[i]++; // !
    }
}
```

Listing 4.2: Undefined behaviour (1): Access outside specified memory region

```
int x[10];

#pragma oss task reduction(+: x)
{
    int i = random(0, 9);
    x[i]++;

    int y = x[0]; // !

    for (int i = 0; i < 10; ++i)
        printf("x[i] = %d\n", x[i]); // !
}
```

Listing 4.3: Undefined behaviour (2): Reading partial results

```
int x[10];

#pragma oss task reduction(+: x)
{
    for (int i = 0; i < 10; ++i)
    {
        x[i] *= i; // !
    }
}
```

Listing 4.4: Undefined behaviour (3): Incompatible operators

```
int A[100];
int *Aptr = &A[10];

#pragma oss task reduction(+: A) \
    firstprivate(Aptr)
{
    A[10]++;
    (*Aptr)++; // !
}
```

Listing 4.5: Undefined behaviour (4): Memory consistency problems

```
int A[150];
int *ptr1 = &A[0], *ptr2 = &A[50];

#pragma oss task \
    reduction(+: [100]ptr1) \
    reduction(*:  [100]ptr2)
{
    for (int i = 0; i < 100; ++i) {
        ptr1[i]++; // ! A[0:99]
        ptr2[i] *= 2; // ! A[50:149]
    }
}
```

Listing 4.6: Undefined behaviour (5): Reduction principle not fulfilled

```
int A[150];
int *ptr1 = &A[0], *ptr2 = &A[50];

#pragma oss task \
    reduction(+: [100]ptr1) \ // !
    reduction(*:  [100]ptr2) // !
{
    for (int i = 0; i < 50; ++i) {
        ptr1[i]++; // A[0:49]
        ptr2[i] *= 2; // A[50:99]
    }
}
```

Listing 4.7: Undefined behaviour (6): Badly annotated code

```
int A[150];
int *ptr1 = &A[0], *ptr2 = &A[50];
#pragma oss task reduction(+: [100]ptr1) \
    reduction(+:  [100]ptr2)
{
    for (int i = 0; i < 100; ++i) {
        ptr1[i]++; // A[0:99]
        ptr2[i] += 2; // A[50:149]
    }
}
```

Listing 4.8: Overlapping reductions within the same task

In addition, any access to the memory referenced by a reduction variable through another variable not specified in a `reduction` clause will induce an undefined behaviour (code 4.5), since the consistency between the privatised memory and the original variable cannot be guaranteed under those circumstances. When the variable is specified in a `reduction` clause but the reduction operators do not match, either the reduction principle is unsatisfied (operations over the data are not associative and commutative, code 4.6), or the task is badly annotated (code 4.7). Both situations derive to undefined behaviour. On the contrary, two compatible reduction variables overlapping in memory can be handled as two completely independent reductions, as shown in the code listing 4.8.

4.3 The dependence-data sharing duality of the `reduction` clause

As soon as privatisation becomes necessary, the `reduction` clause stops behaving as a sole dependence and starts assuming responsibilities of a data sharing attribute. As this happens, the following dualism is presented:

On the one hand, the dependence component of the `reduction` clause has an effect on the memory reference (*lvalue*) that corresponds to the specified variable. On the other hand, the data sharing attributes in *OmpSs-2* can only be defined on symbols. Data sharing attributes defined over memory references would be confined to highly restricted situations.

As an intuition, imagine a section of an array that is `private` within the task, whereas another section of the same array is `shared`, and both are accessed through the same symbol `x`. Now, imagine a function is called over `x`, as illustrated in the code listing 4.9. A symbol can only reference a memory region, hence we can only define a single data sharing attribute for it. Reductions present the same problem, and are the only dependence type in *OmpSs-2* to have it.

```
#pragma oss task private(x[0:4]) shared(x[5:9]
{
    // There is only one symbol for variable x
    // so there can only be one data sharing attribute
    foo(x);
}
```

Listing 4.9: Multiple data sharing clauses over the same symbol (incorrect)

This duality imposes some additional restrictions on the use of the `reduction` clause. First, when a symbol appears in a `reduction` clause for a task, it must not appear in any other dependence clause of the same task, as those imply a different data sharing attribute (see section 1.3.3). Furthermore, specifying the same symbol on multiple `reduction` clauses on the task could result in having a reduction over a discontinuous memory region, which is referenced only by a single symbol (the variable).

In this situation, depicted in the code listing 4.10, the only solution would be to privatise the whole memory between the two discontinuous regions. However, this can be very inefficient if those are far separated. Instead, we propose forwarding this responsibility to the user, who can easily install a pointer in order to express such discontinuities without compromising efficiency, as shown in the code listing 4.11. For this reason, expressions like `Array Sections` are not allowed in the `reduction` clause.

```
int *hugeArray = malloc(1024*1024*1024*sizeof(int));

// ! Array Sections are not permitted in the reduction clause
#pragma oss task \
    reduction(+: hugeArray[0; 256]) \
    reduction(+: hugeArray[1024*1024*1024 - 256; 256])
{
    computeHead(hugeArray, 256);
    computeTail(&hugeArray[1024*1024*1024 - 256], 256);
}
```

Listing 4.10: Discontinuous array reduction (incorrect)

```
int *hugeArray = malloc(1024*1024*1024*sizeof(int));
int *arrayHead = hugeArray;
int *arrayTail = &hugeArray[1024*1024*1024 - 256];

// Shaping Expressions are allowed in the reduction clause
#pragma oss task reduction(+: [256]arrayHead, [256]arrayTail)
{
    computeHead(arrayHead, 256);
    computeTail(arrayTail, 256);
}
```

Listing 4.11: Discontinuous array reduction (correct)

4.4 Overlapping array reductions and partial combination

The *OmpSs-2* programming model has been designed around the data-flow concept.[5] Consequently, the array reductions mechanism has also been driven by it.

As explained in section 1.3.2.2, *OmpSs-2* dependences are precisely computed by figuring out the exact intersection between the memory references annotated in the different tasks dependence clauses. This feature allows the efficient execution of patterns where tasks dependences have partial overlaps (that is, their dependences overlap, but not match completely), or nested programs where tasks are divided into smaller subtasks, such as the *mergesort* displayed in the code listing 4.12. Figure 4.1 shows the task dependence diagram, where `sort` tasks are represented in yellow whereas `merge` tasks are represented in green.

```
void sort(int n, int *a) {
    if (n < threshold)
        seq_sort(n, a);

    #pragma oss task inout(a[0:(n/2)-1])
    sort(n/2, a);

    #pragma oss task inout(a[n/2:n-1])
    sort(n/2, &a[n/2]);

    #pragma oss task inout([n]a)
    merge(n/2, a, &a[0], &a[n/2]);
}
```

Listing 4.12: Mergesort using *OmpSs-2* fine-grained dependences

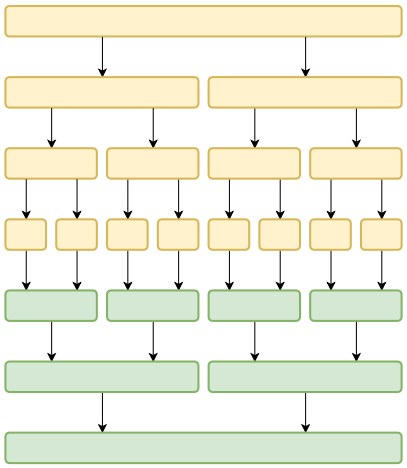


Figure 4.1: Mergesort task dependence diagram

In the same fashion, the reduction mechanism is expected to support patterns where the task dependences are fragmented and overlapped. Code listing 4.13 shows a synthetic example used for illustration purposes. Its task dependence diagram is shown in figure 4.2. In this example, each input task that depends on the overlapped reduction tasks should trigger the combination of the reduction for the specific memory region they will access, but not necessarily for any other region. For instance, the second input task in the diagram will require both the first and the second reduction tasks to have completed their execution, and `data[BS:2*BS]` to be combined, but not necessarily `data[0:BS]` or `data[2*BS:3*BS]`. We refer to this concept as *partial combination*, and it is fundamental to provide the flexibility required to compose with the other components in the programming model.

```
double *data = ...

for (int i = 0; i < N - 1; i += BS) {
    double *ptr = &data[i]
    #pragma oss task reduction(+: [2*BS]ptr)
    {
        for (int ii = 0; ii < i + 2*BS; ++i) {
            [...]
        }
    }
}

for (int i = 0; i < N; i += BS) {
#pragma oss task in(data[i; BS])
{
    for (int ii = 0; ii < i + BS; ++i) {
        [...]
    }
}
```

Listing 4.13: Task reduction with partial overlap

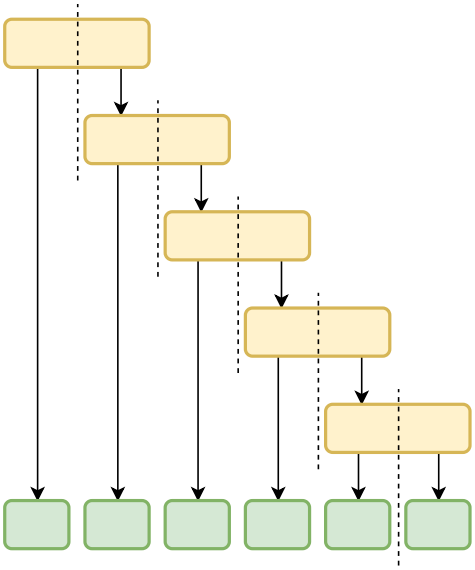


Figure 4.2: Task reduction with partial overlap dependence diagram

4.5 Nesting model and the `weakreduction` clause

As with any other dependence in *OmpSs-2*, reduction tasks can be nested. Regardless of the chosen strategy, nested reductions can be hindered by the privatisation process.

For instance, in situations that involve the compilation of more that one *compilation unit*, such as the one shown in the code listing 4.14, it can become a problem to detect that a reduction is registered over a private storage. In those circumstances, supporting nesting directly can add a significance performance penalty, as explained in sections 2.2.1 and 2.2.2. For this reason, it was decided that the nesting support would be provided through a different mechanism: the `weakreduction` clause.

The `weakreduction` clause is the `weak` counterpart of the `reduction` clause. Analogous to the other weak-dependence clauses (described in section 1.3.2.1), the `weakreduction` clause is used to state that the reduction over the data will not be carried out by the task, but by some nested subtask instead. Since the task annotated with the `weakreduction` will not be accessing the data directly, the privatisation is not necessary at that point. Finally, avoiding the privatisation allows an efficient nesting, equivalent to that of any other dependence clause. With it, the real reduction computation will be performed at the last level of nesting, which will be annotated using the

strong `reduction` clause. Code listing 4.15 shows a recursive dot-product using the `weakreduction` mechanism.

```
void foo(int n, int *x) {
    #pragma oss task \
        reduction(+: [n]x)
    {
        bar(&x, n);
    }
}
void bar(int *data, n) {
    #pragma oss task \
        reduction(+: [n]data)
    {
        [...]
    }
}
```

Listing 4.14: Nested reductions in separate compile units

```
void dot_product(int n, int &res,
    int *a, int *b)
{
    if (n < threshold) {
        #pragma oss task \
            reduction(+: res)
        seq_dot_product(n, res, a, b);
    }

    #pragma oss task \
        weakreduction(+: res)
    dot_product(n/2, &a[n/2], &b[n/2]);
}
```

Listing 4.15: Nested reductions using `weakreduction`

5. Reference implementation of the proposed extension

After having described the proposed extension of the programming model from a formal point of view, this chapter introduces a base implementation of it based on the reference *OmpSs-2* infrastructure, that is composed by the *Mercurium* source-to-source compiler and the *Nanos6* runtime support library. In the following sections, the most relevant implementation decisions will be discussed in detail, while pointing out the principal design characteristics.

Our complete base implementation as described in this chapter plus the optimisations presented in chapter 6 will be published in the *Mercurium*[1] and *Nanos6*[2] code repositories, as well as distributed in the next release of the *OmpSs-2*[3] programming model, which is expected by June 2019.

5.1 Choosing a privatisation mechanism

Our implementation was designed having our minds set in dense array reductions where most available CPUs participate. We believe this is the most challenging scenario to efficiently parallelise by using the currently available mechanisms in the *OmpSs-2* programming model, and thus the one that can be most improved. Many of the implementation choices described in this chapter are determined by this idea.

An important decision taken during the initial design phase of our implementation was to decide whether our design would rely on a privatisation mechanism or not and, if so, which would this mechanism be.

If we refer back to the synchronisation-based parallelisation strategies described back in section 2.1.1, we will notice how the serialisation of tasks is already implemented in *OmpSs-2* by using the `inout` clause or `commutative` clauses (defined in section 1.3.2).

[1] `https://github.com/bsc-pm/mcxx`
[2] `https://github.com/bsc-pm/nanos6`
[3] `https://pm.bsc.es/ompss-2-downloads`

On the other hand, explicit synchronisation strategies using *locks* or atomic instructions are not enough to enough to completely fulfil the proposed *OmpSs-2* specification extension, as briefly explained in section 4.2.

In an attempt to provide a better grasp on how this limitation affects an actual implementation of the programming model, we need to understand how the compiler transforms the original code of a reduction task in order to generate an equivalent code that uses an explicit synchronisation mechanism.

For the particular case of atomic instructions, this limitation originates from aspiring to exploit the benefits they offer without the user having to make modifications to the task code. This demand requires the compiler to traverse the task code, replacing any reference to the reduction expression defined in the `oss reduction` clause by an atomic access to the data. This transformation is exemplified seen in code listings 5.1 and 5.2.

```
#pragma oss task reduction(+: x)
{
    x++;
}

#pragma oss task reduction(*: y)
{
    y *= 2;
}
```

Listing 5.1: Taskified reductions using *OmpSs-2*

```
void mcxx_outlined_task_0(int *x) {
    atomic_fetch_and_add(x, 1);
}

void mcxx_outlined_task_1(int *y) {
    do {
        int y_old = __atomic_load(y);
        int y_aux = y_old*2;
    }
    while (!atomic_compare_and_swap(
                y, &y_old, y_aux));
}
```

Listing 5.2: Transformed code by the *Mercurium* compiler

Even if this transformation may seem straightforward in the simple cases, the compiler is only able to perform such substitution when all code is visible, which unfortunately is not always the case. Code listing 5.3 shows an example where the reduction task calls an external function defined in a library, which would make this transformation fail (accesses to the memory region inside the library would not be consistent). When the visibility requirements are met, the *OmpSs-2* `concurrent` clause can be used to implement a user-driven parallelisation of reduction patterns using atomic instructions.

On the other hand, the contention problems brought by a *lock*-based synchronisation strategy discussed in section 2.1.1 render that variant unsuitable for the applications we wish to optimise (dense reductions where most CPUs participate). At this point,

```
#pragma oss task in(X) \
    reduction(+: [N]Y)
{
    cblas_daxpy(N, 4 , X, 1, Y, 1);
}
```

Listing 5.3: Taskified reduction with externall function call

```
void mcxx_outlined_task_2(int (*Y)[N])
{
    int _size = N*sizeof(int);
    int (*Y_aux)[N] =
        nanos6_get_prv_storage(Y, _size);

    cblas_daxpy(N, 4 , X, 1, *Y_aux, 1);
}
```

Listing 5.4: *Mercurium* transformation enabling CPU-privatisation (simplified)

it is clear that a privatisation strategy is required to extract the inherent parallelism from the reduction pattern while fulfilling the *OmpSs-2* programming model.

Even though the task-privatisation strategy would fulfil the specification requirements and thus is correct, we have to consider its efficiency and impact on memory. Taking into account that the number of tasks in an *OmpSs-2* program is unbounded and that the purpose of this extension is to support reductions of arrays; the cost of having to allocate, initialise and combine a new private storage for each task would exceed the benefits, making the mechanism ineffective.

In order to avoid the previous problems, we finally opted for the CPU-privatisation strategy, in which a private storage is allocated for every CPU participating in the reduction and those are orchestrated from within the runtime. The main idea behind this strategy is that, given that only as many tasks as CPUs can be executed simultaneously, having as many private storages as CPUs should be enough to execute all reduction tasks without any storage being used by more than one CPU.

Furthermore, having the private storage bound to a CPU allows it being reused for the execution of different reduction tasks. Contrary to task privatisation, this reuse allows the allocation, initialisation and combination costs of a given private storage to be split among all tasks that have used it.

Besides, as soon as the private storages are managed by the runtime, so is their combination. With this, synchronisation constructs that were required in task-privatisation can be replaced by a runtime-controlled combination mechanism. This mechanism ensures that ensures memory is kept consistent. Code listing 5.4 shows the compiler transformation that enables a CPU-privatisation in the previous example. In this transformation, *Mercurium* inserts a call to the *Nanos6* runtime that will be used to obtain the CPU-private storage and replaces all references to the original reduction symbol for the obtained private storage.

Finally, in our implementation reduction tasks are bound to a single CPU. This means that an arbitrary (strong) reduction task will be wholly executed in the CPU where it has begun executing. This requirement is set to deal with the concern of task migration between CPUs, described in section 2.1.2.2. In it, reduction tasks can end up using the incorrect private storage by migrating to another CPU without updating it.

5.2 Supporting overlapping reductions

Considering two reduction tasks defined with the same reduction operator. When the data specified in the first reduction task overlaps with the data specified by the other, those reductions are said to *overlap*. In section 4.4, the proposed specification extension states that overlapping reductions are valid and need to be supported.

Within overlapping reductions, there are two possible types of overlap: There is either a full overlap where a region is fully contained within the other or a partial overlap, where the contrary happens. Both types are exemplified in code listing 5.5 and figure 5.1.

```
int A[100];
int *ptr = &A[25];

// Full overlap
#pragma oss task reduction(+: [100]A)
{
}

#pragma oss task reduction(+: [50]ptr)
{
}

// Partial overlap
#pragma oss task reduction(+: [50]A)
{
}

#pragma oss task reduction(+: [50]ptr)
{
}
```

Listing 5.5: Types of reduction overlaps

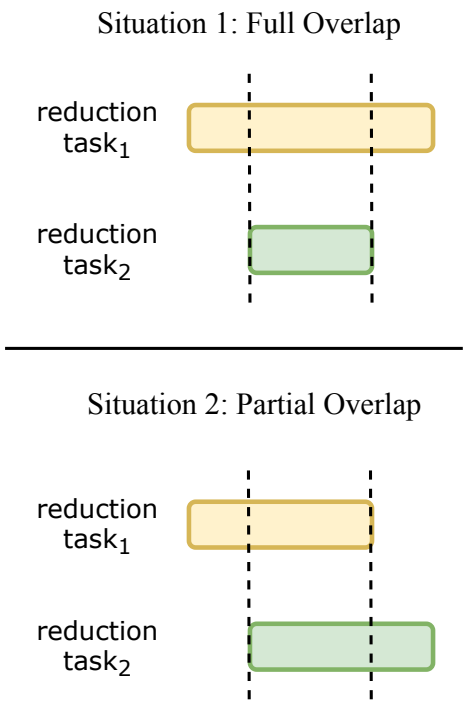


Figure 5.1: Graphical representation of overlap types

Complete overlaps where the first registered reduction contains the second are the

simplest scenario. In our implementation, this scenario is supported by allowing the second reduction to use the private storages that are already allocated for the first. With this, a correct computation is guaranteed while we avoid a new privatisation set for the second task.

In contrast, full overlaps in the opposite order and partial overlaps are managed in the same manner. In those scenarios, the privatisation of the first task does not meet the required privatisation needs for the second. At this point we can choose between two different mechanisms:

- Relocate first privatisation and enlarge it to fit the second task privatisation
- Allocate a new set of private storages just for the second task

(Note that a strategy where the overlapping part is reused from the existing privatisation while allocating the remaining memory is not possible given the dependence/data sharing duality)

In the *relocation* strategy, the total amount of consumed memory is minimised at the expense of having to deal with memory movements. Moreover, the relocation process is not trivial, as the first reduction task may have already started and the private storages may be already in use.

On the other hand, allocating a new set of private storages prioritises speed over memory and deals with the second task as if it was part of an independent reduction. For its simplicity and increased performance, this was the chosen mechanism in our implementation.

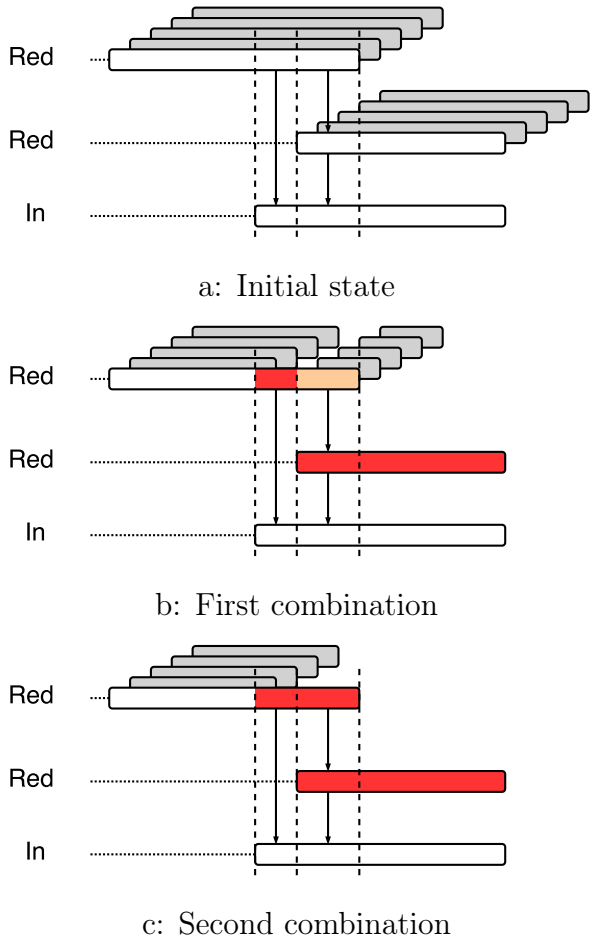
In overlapping reductions, as in any other regular reduction, the combination into the original data can be done as soon as all tasks belonging to the reduction have been executed. However, if there are multiple sets of private storages for the same data (the overlapped region), an ordered combination of the different sets is required.

The code listing 5.6 shows two tasks participating in a (partially-overlapping) reduction followed by a third task that has an input over a part of the reduction data. In this code, the reduction will be combined in two steps: All regions that do not overlap can be combined in a first step, in conjunction with the overlapping region in one of the privatisation sets. Then, the overlapping region in the second privatisation set will be combined at a second step. Figure 5.2 shows this process graphically.

Nevertheless, a mechanism capable of dealing with this situation more efficiently is presented in section 5.3.

```
int data[100] = ...
int *ptrA = &data[0];
int *ptrB = &data[50];

#pragma oss task \
    reduction(+: [60]ptrA)
{
}

#pragma oss task \
    reduction(+: [50]ptrB)
{
}

#pragma oss task \
    in(data[40:99])
{
}

#pragma oss taskwait
```

Listing 5.6: Partially overlapping reduction



a: Initial state

b: First combination

c: Second combination

Figure 5.2: Orderded combination of partially overlapping reductions

5.3 Supporting nested reductions

The *OmpSs-2* nesting model is defined in the section 4.5 of the specification proposal. There, the `weakreduction` clause is presented as an effective solution to support nested reductions without having to cope with the inefficiencies derived from privatisation.

Tasks annotated with `weakreduction` clauses are not supposed to access the data directly, and thus no privatisation is performed. Without privatisation, there is no need for the compiler to replace any reference to the reduction variable, hence the `weakreduction` dependence type is handled by the runtime as any other dependence type (the dependence/data sharing duality vanishes).

In our implementation the `weakreduction` clause is used to preallocate contiguous storages, anticipating nested overlaps and hence reducing unnecessary privatisations. The idea behind this claim is the following:

If a memory region appears in a `weakreduction` clause of a task, it means that its subtasks will be accessing that memory region at some point. In this situation, delaying the privatisation until the registration of reduction subtasks is a legit strategy. However, it will inevitably result in extra privatisations when there are overlaps between the reduction subtasks, as seen in section 5.2.

In contrast, the information provided in the `weakreduction` clause can be used as a hint to foresee the privatisation needs of the reduction subtasks. That is, by privatising the whole memory region, any private storages required by the nested tasks is guaranteed to fit into it, and therefore the extra allocations can be avoided. Code listing 5.7 exemplifies this situation.

```
int data[100] = ...
int *ptrA = &data[0];
int *ptrB = &data[50];

#pragma oss task \
    weakreduction(+: [100]data)
{
    #pragma oss task \
        reduction(+: [60]ptrA)
    {
    }

    #pragma oss task \
        reduction(+: [50]ptrB)
    {
    }
}

#pragma oss taskwait
```

Listing 5.7: `weakreduction` with overlapping subtasks

6. Optimising array reductions in *OmpSs-2*

Following the description of the reference implementation of the proposed extension, this section details a set of optimisations that have been developed on top of it.

In the following subsections, each optimisation will be explained separately, providing, for each, a motivational use-case and comparing it to the base implementation.

It is noteworthy to point out that none of the presented optimisations should change the expected behaviour defined by the programming model in section 4 in any sense. Those are only intended to speed-up some common reduction patterns that were frequently found in the analysed applications. Any program that complies with the specification should remain correct after applying any combination of these optimisations.

6.1 Padding the private storage

This optimisation is the most simple of the proposed, yet it can have a great impact on the execution time.

This optimisation aims to eliminate the false sharing coherence artefact. In short, false sharing can be defined as a memory pattern in coherent cache based systems where processors in a shared-memory parallel system make references to *different* data objects within the same coherence block (cache line or page), thereby inducing "unnecessary" coherence operations.[27] These coherence operations usually consist in invalidating the cache line and reloading it, and can vastly degrade an application performance.

As far as this work is concerned, false sharing may appear when a reduction is performed over a memory region that is smaller than the cache line. Scalar reductions are a particular yet very relevant situation where this condition is met.

The optimisation itself consists in detecting when a private reduction storage is susceptible to false sharing (when smaller than the cache line), and place the necessary

padding bytes up to cache-line so that no other private reduction storage falls in such cache line. Figure 6.1 shows a graphical representation of this process.

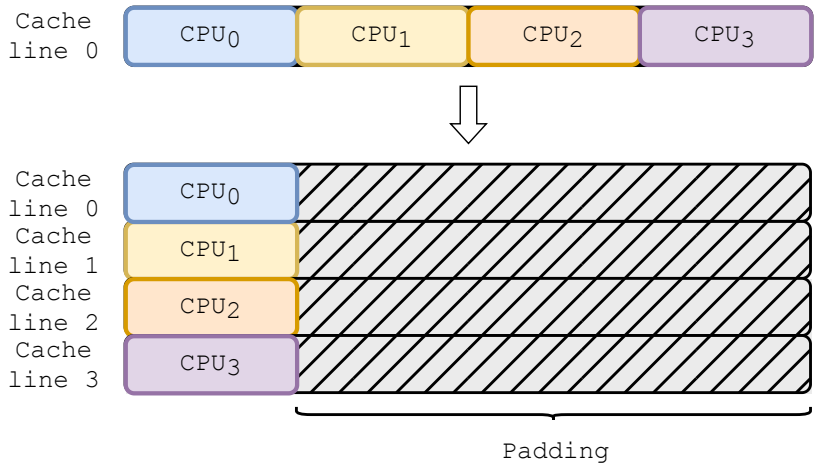


Figure 6.1: Add *padding* to cache lines to avoid falsesharing

6.2 Early beginning

The *Early beginning* optimisation consists in making reduction tasks begin their execution before their dependences are satisfied. This is possible thanks to the privatisation mechanism, which allows the reduction tasks to exclusively use private storages during their execution. The original memory region is only required to be consistent and to have satisfied any previous dependences by the end of the pattern computation, when all private storages are to be combined into it.

This optimisation is useful to increase resource occupancy and to reduce load imbalance: It makes reduction tasks ready to be executed as soon as they are created (assuming they satisfy the other dependences).

6.2.1 Use case: Heat diffusion solver

The proposed use case for this optimisation is a code that simulates heat diffusion in a solid body using a *Gauss-Seidel*[1] solver for the heat equation. The algorithm takes some heat sources, their position, size, and temperature and the dimensions of the solid body. In turn, it computes the resulting heat distribution over the solid body and generates an image displaying the temperature gradient from red (hot) to dark blue (cold). Figure 6.2 shows the resulting distribution of a sample execution with two heat sources placed on the edged of a 2D solid.

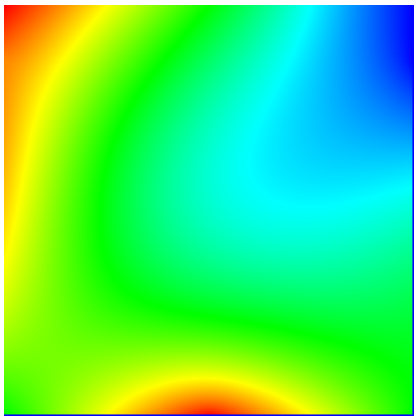


Figure 6.2: Resulting distribution of a sample execution with two heat sources

The *Gauss Seidel* solver computes the temperature gradient by iteratively updating the gradient matrix until convergence is reached (a residual is matched) or the maximum number of iterations have been computed.

Within each iteration, the *Gauss-Seidel* is decomposed in blocks that are then taskified in *OmpSs-2*. The block dependence schema is shown in figure 6.3. As we can see in the previous figure, two blocks can only be executed simultaneously if their dependences are both met, and this will only happen when the blocks are found in the same counter-diagonal. As a consequence, the maximum parallelism achievable with this application can be calculated as the number of blocks in the longest counter-diagonal. In short, the data flow of this solver can be seen as a wave-front that starts in the upper left block and propagates throughout the counter-diagonals until the lower-right block. Finally, the computation of each block contributes to the computation of the residual by means of an addition. This pattern can be modelled as a reduction in *OmpSs-2*.

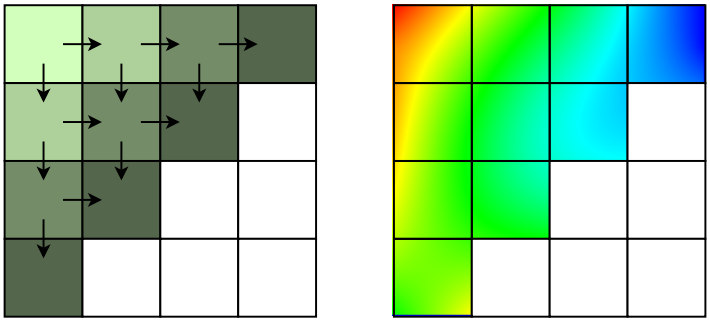


Figure 6.3: *Gauss-Seidel* task parallelization in *OmpSs-2*

After each iteration, all tasks are waited for in a `taskwait` directive in order to materialise the residual computation (into a `sum` accumulator). The residual is then used to assess whether the problem has converged, or if another iteration is necessary otherwise. Subsequently, the heat diffusion phases extracted from the code listing 6.1 are

represented in the diagram shown in figure 6.4.

```
double sum = 0.0f;

for (int t = 0; t < timesteps; ++t) {
    solveGaussSeidel(
        matrix, rowBlocks, colBlocks, sum);

    #pragma oss taskwait in(sum)
    if (sum < delta)
        break;
}

return sum;
```

Listing 6.1: *Gauss-Seidel* residual check

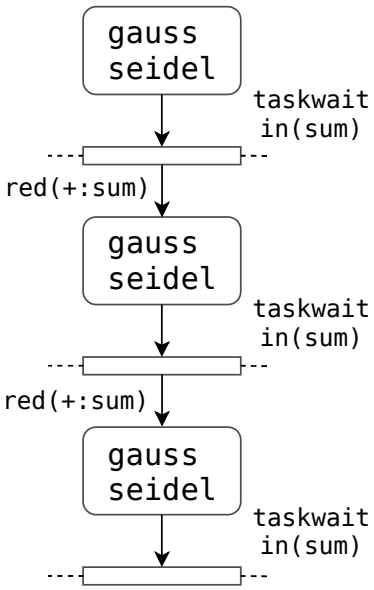


Figure 6.4: *Gauss-Seidel* phase diagram

What we want to show with this mini-app is that the *early beginning* optimisation can help boost the block parallelism by overlapping two wave-fronts (*Gauss-Seidel* iterations) in time. To do so, the taskification of the algorithm has to be slightly changed by introducing an auxiliary `old_sum` variable. This variable is used to hold the residual of the first ongoing wave-front, while the `sum` variable will hold the residual for the second wave-front. Consequently, the `taskwait` will now wait for the `old_sum` variable, which will be updated from the `sum` variable by means of a task. The code listing 6.2 and the diagram in figure 6.5 show the modified code structure.

With this change, the thread creating the tasks will instantiate the tasks necessary for computing two complete wave-fronts before stopping in a `taskwait` for the first time and, from that point onwards, will always instantiate the tasks one iteration *ahead* of the oldest wave-front currently being executed. In this scenario, the *Early Beginning* optimisation relaxes the output dependence over the `sum` variable, allowing that tasks from the *next iteration Gauss-Seidel* begin their execution before the previous iteration and the `old_sum` update task are completed. Figure 6.6b shows the regular execution of a wave-front at a time, whereas figure 6.6b shows how *Early Beginning* enables two wave-fronts to be simultaneously executed (pipelined).

```
double sum = 0.0f;
double old_sum = delta;

for (int t = 0; t < timesteps; ++t) {
    solveGaussSeidel(
        matrix, rowBlocks, colBlocks, sum);

    #pragma oss taskwait in(old_sum)
    if (old_sum < delta)
    break;

    #pragma oss task label(update) \
        inout(sum) out(old_sum)
    {
        old_sum = sum;
        sum = 0.0f;
    }
}

#pragma oss taskwait
return sum;
```

Listing 6.2: *Gauss-Seidel* modified residual check

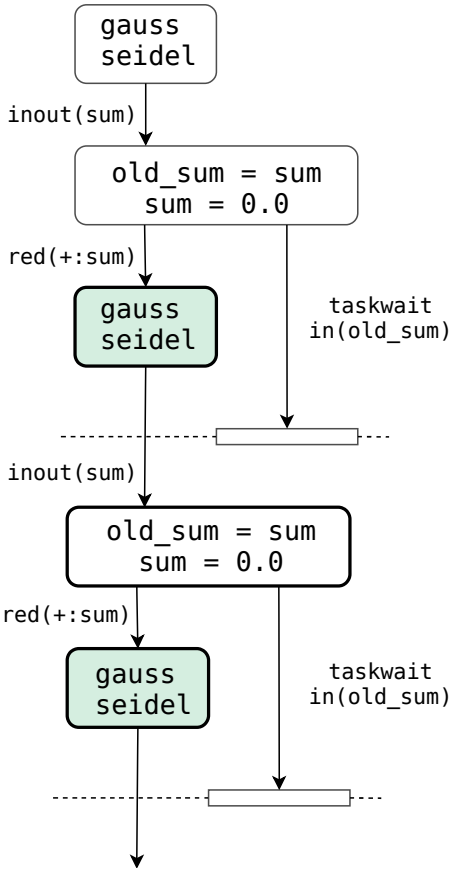
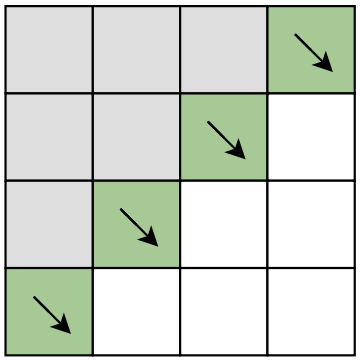
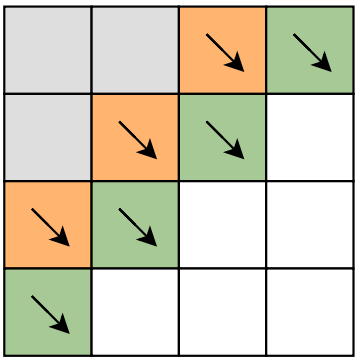


Figure 6.5: *Gauss-Seidel* modified phase diagram

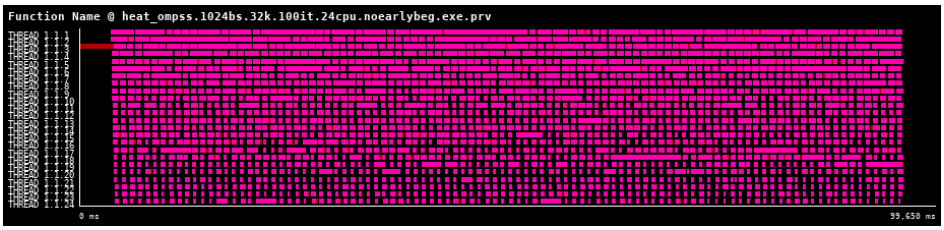


(a) A single wave-front is executed at a time

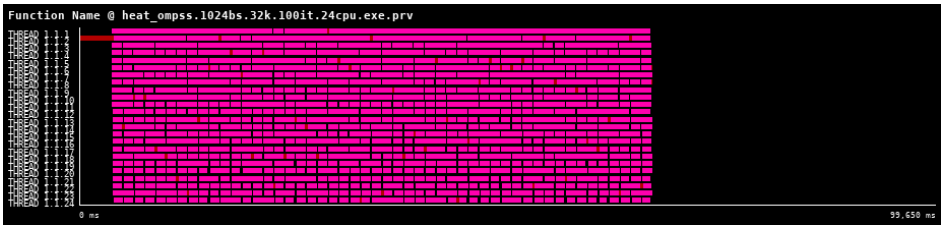


(b) *Early Beginning* allows two wave-fronts to be simultaneously executed

This pipeline magnifies the useful parallelism and, with it, the effective resource usage, therefore the overall execution time is reduced. The following figures show *Paraver* traces where we can see the effect that the *Early Beginning* optimisation (6.7b) has on the execution time compared to the base version (6.7a).



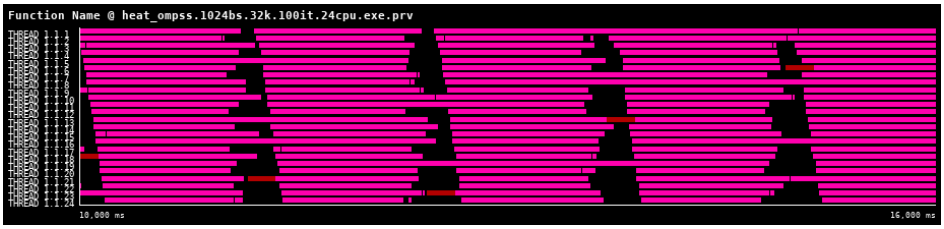
(a) Heat diffusion trace using *Gauss-Seidel* solver (base)



(b) Heat diffusion trace using *Gauss-Seidel* solver (*Early Beginning*)

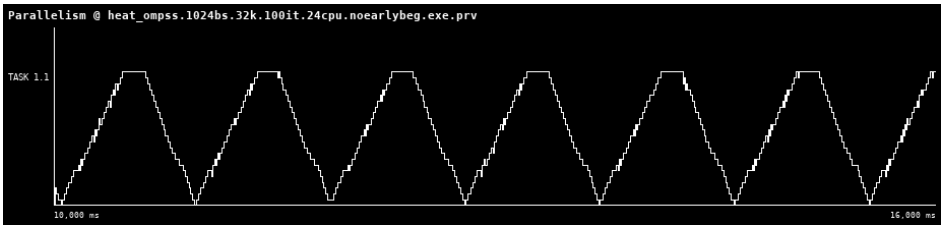
In figure 6.8a, we now zoom into the previous trace and focus on a few *Gauss-Seidel* iterations only. We can clearly see the shape of the wave-front within each iteration. In particular, we can see a slope in the vertical axis. This slope corresponds to the delay caused by having dependences between blocks. Figure 6.8b shows the *Early Beginning* counterpart, where each *block* in the trace corresponds to *two* pipelined *Gauss-Seidel* iterations. We can see how the extra parallelism provided by the optimisation is used to increase the resource usage and shorten the mean execution time per iteration.

(a) 6 complete *Gauss-Seidel* iterations (base, zoom)

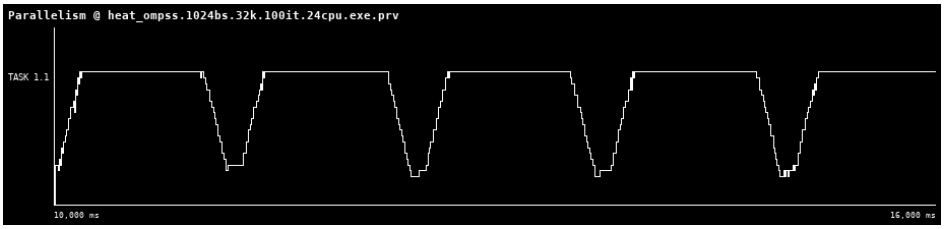


(b) 10 complete *Gauss-Seidel* iterations (*Early Beginning*, zoom)

Finally, figures 6.9a and 6.9b show the parallel efficiency at this zoomed part of the trace. In the base execution, the maximum efficiency is just reached in a small fraction of each iteration (when all blocks in the main counter-diagonal are being executed), while with the *Early Beginning* optimisation, this fraction has been considerably increased.



(a) Heat diffusion parallel efficiency using (base, zoom)



(b) Heat diffusion parallel efficiency using (*Early Beginning*, zoom)

6.3 Dynamic privatisation strategy

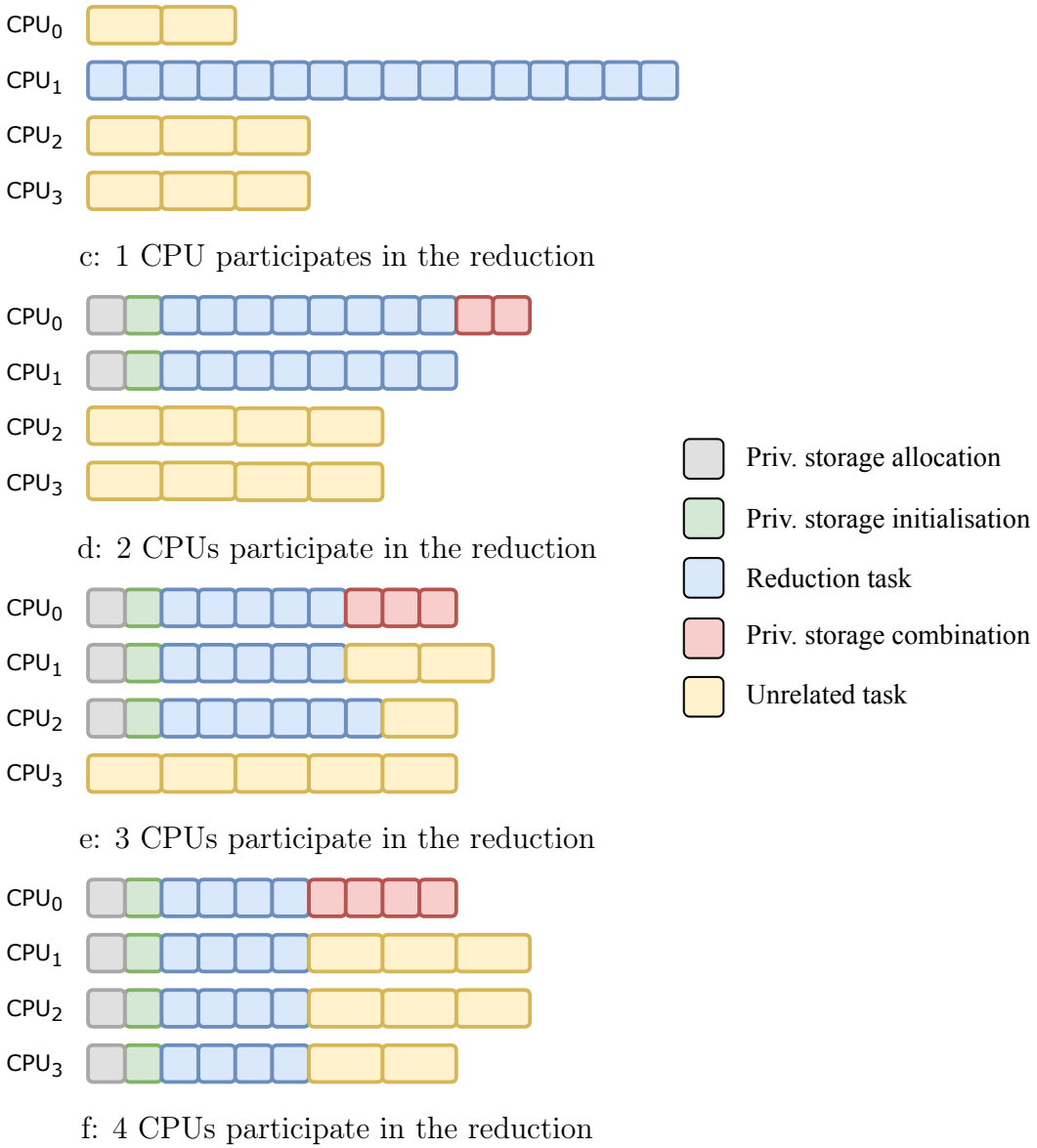
As discussed in section 5.1, the CPU-privatisation strategy is optimised for scenarios where all CPUs participate in reduction the process (that is, all CPUs execute at least one task belonging to the same reduction). However, other scenarios may require a more flexible privatisation mechanism. In this section a *dynamic* CPU-based privatisation is presented.

In our original CPU-privatisation implementation, all private storages are allocated as soon as the reduction dependences are registered, when the privatisation needs are revealed. Then, each of the private storages is assigned to a single CPU, and that bound is held until the finalisation of the reduction.

In this strategy, the cost of privatisation (allocation, initialisation, combination, etc.) is independent of the number of tasks in the reduction. Instead, it depends solely on the number of CPUs available in the system. Besides, we tend to think that the more CPUs participate in the reduction, the faster its computation will be. However, depending on the problem to solve and its parallelisation, this is not always true.

Figure 6.10 aims to give some intuition on this by showing a small example. In it, we can see how a reduction composed of 16 tasks is computed using from one to four CPUs. In addition to the execution of the reduction tasks and the processes derived from privatisation, the figures show the execution of eight extra tasks. Those tasks are independent and do not belong to the reduction, they are only used to illustrate the occupancy of the resources.

As we can see, the latency of computing the reduction in the example is the same when using three (6.9e) or four (6.9f) CPUs. However, the extra resource used in the latter forces other tasks to be delayed, resulting in a longer execution.



c: 1 CPU participates in the reduction

d: 2 CPUs participate in the reduction

e: 3 CPUs participate in the reduction

f: 4 CPUs participate in the reduction

Figure 6.10: Reduction latency when different number of CPUs participate

A clear example of a situation where adding CPUs does not help is perhaps a memory-bounded application. In such applications, the maximum bandwidth of the system can be reached using a relatively small number of CPUs. At this point of saturation, it is pointless to increase the number of CPUs participating in the reduction, as the effective bandwidth will be divided among the CPUs providing no real benefit. Instead, we will be wasting resources that could be otherwise executing other compute-bounded tasks, or even be halted to increase energy efficiency.

For the previous reasons, we believe that it is necessary to have mechanisms able to control the amount of resources assigned to the computation of a reduction. Dynamic resource management is a complete work in its own, and its application on task reductions is clearly beyond the scope of this project. However, we have decided to make some steps towards that direction by implementing a dynamic CPU-based privatisation strategy.

In this strategy, no preallocation is performed when a reduction is registered. Instead, we create an empty pool of private storages. Then, when a reduction task is to be executed, a new private storage is allocated and initialised as usual. When the execution of that task is over, the CPU *releases* the private storage into the pool.

In general, when a CPU is to execute a task, two situations can happen: On the one hand, there may be unused storages in the pool, in which case the CPU will use one of them to execute the task and save a privatisation. On the other hand, the pool may have no available storages. When this happens, the CPU will allocate a new storage, which will also be released onto the pool after the task execution, incrementing the number of private storages by one.

Using this mechanism, we ensure that private storages will only be allocated when they provide useful parallelism. In other words, the number of private storages in the pool will correspond to the maximum parallelism achieved for the computation of that reduction. As a consequence, the pool will hold at most as many private storages as CPUs.

While the proposed privatisation does not directly solve the presented problem, its flexibility opens the door to future control policies to do it. Moreover, it allows some particular scenarios to be optimised.

For instance, imagine an application that presents a low-priority reduction that is executed only when no other tasks are available. In this situation, the presented optimisation would allow the reduction to be computed using a small number of private storages to be reused among all CPUs, whereas our original strategy would have required a different storage for each CPU.

On the downside, when storages are reused between different CPUs, the data affinity is reduced. For this reason, this privatisation may have an impact on memory access efficiency.

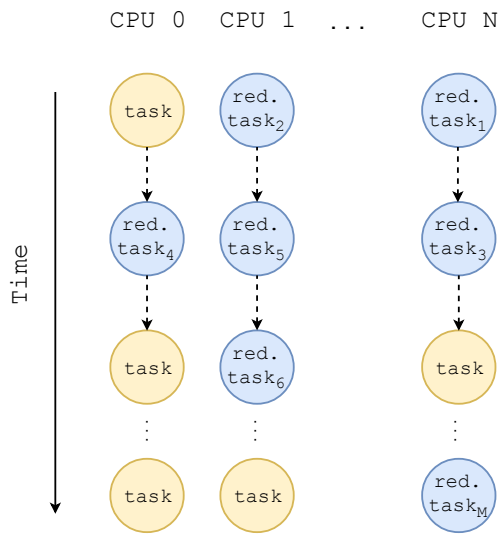
6.4 Original storage reuse

During the computation of a reduction using privatisation, the original storage is only used at the combination phase. The idea behind this optimisation consists in taking advantage of this by reusing the original memory as the private storage for one of the CPUs.

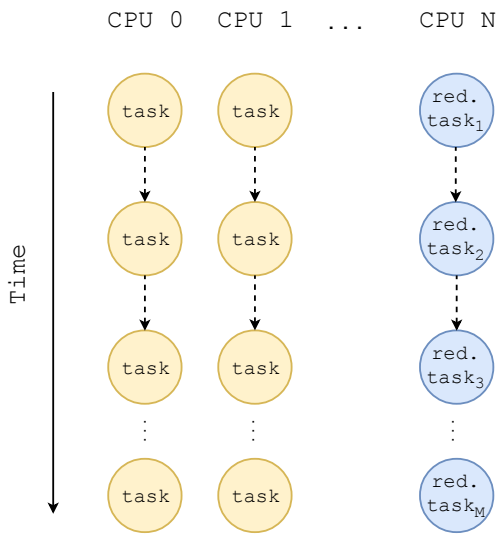
Saving the privatisation of a single storage will not probably have a very big impact on reductions where all CPUs participate. However, it can make a difference in programs where there is already enough parallelism for the available resources provided, for instance, by other tasks that are not part of the reduction. In situations such as this, the task scheduler in the runtime library can decide to schedule reduction tasks one after the other, so that they can all be executed over the original memory.

In the general case, this optimisation allows sparing one allocation, one initialisation and one combination. However, in situations such as the aforementioned, can mean eliminating the reduction overheads when we are not able to take advantage of its benefits.

Figure 6.11a shows a situation where the extra parallelism provided by the reduction pattern can be absorbed by the available resources (privatisation is required), whereas figure 6.11b shows the opposite situation. In it, the program provides enough parallelism so that all resources are occupied, hence the reduction can be serialised without any performance implication. In this second scenario, the original memory can be reused to cut down overheads.



(a) Resources are underused, reduction provides parallelism that increases resource occupancy



(b) Resources are saturated with tasks, reduction provided parallelism is not useful, scheduler can serialize it to cut down overheads

6.5 Initialization/combination vectorization

The cost of CPU-privatisation for reductions can be split into three parts: On the one hand, the cost of allocating and accessing the extra required memory, on the other hand, the cost of initialising this memory with the reduction neutral element, and finally the cost of combining the private storage back to the original memory region. As soon as the reductions in a program start to have some noticeable weight, these overheads become far from negligible. This optimisation focuses on minimising the initialisation and combination phases by means of using vectorisation techniques.

We believe vectorisation is highly suitable for those processes, due to they being heavily structured and repetitive. In short, the initialisation of a private reduction storage consists in a loop generated by the *Mercurium* compiler that basically iterates through all elements in a memory region initialising them to the neutral element for the reduction operand. Code listing 6.3 shows the generated function for the `max` reduction operand. Similarly, the reduction private storage combination is implemented as a loop iterating through all elements in the private storage applying the reduction operand over the original memory region, as shown in 6.4 for the `or` operator.

```
void nanos6_reduction_init_0(
    int *priv, int size)
{
  int num_elements = size/sizeof(int);

  for (int i = 0; i < num_elements; ++i)
    priv[i] = SHRT_MIN;
}
```

Listing 6.3: `max` initializer function for `int` type

```
void nanos6_reduction_comb_1(
    short *out, short *in, int size)
{
  int num_elements = size/sizeof(short);

  for (int i = 0; i < num_elements; ++i)
    out[i] |= in[i];
}
```

Listing 6.4: `or` initializer function for `short` type

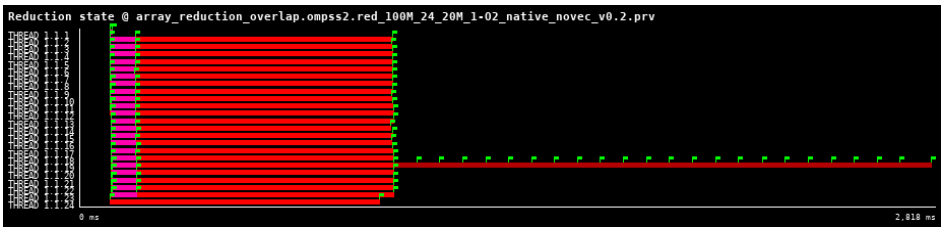
The idea of vectorising the initialiser and combiner routines is to be able to initialise and combine more than one element at once using the Single Instruction Multiple Data (SIMD) Instruction Set Architecture (ISA) extensions.

The number of elements to be computed at once using a SIMD instruction fundamentally depends on the available instructions in the underlying architecture. Generally speaking, the smaller the type size, the greater the number of elements that can be computed at once.

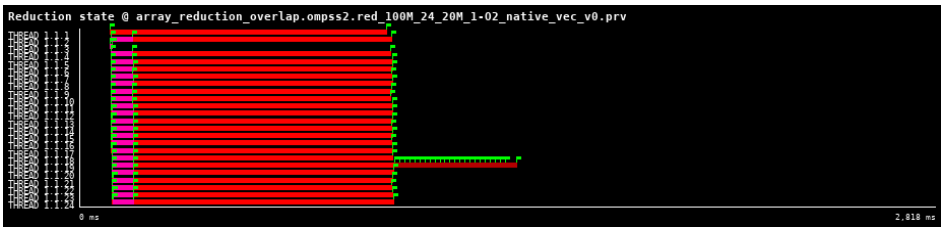
6.5.1 Proof of concept

The Intel architecture found in the *MareNostrum4* supercomputer[7] supports all *MMX* and *SSE* extensions, as well as the newer *AVX*, *AVX2* and *AVX-512*.

In order to exemplify the importance of this optimisation, we have developed a benchmark where we mainly stress the initialisation and combination phases of a reduction. The benchmark consists in instantiating 24 reduction tasks each of which performs 20 million accesses over a memory region of 100 MiB. Considering we run the benchmark on a *MareNostrum4* NUMA-node, 23 extra private storages will need to be allocated, initialised and combined (considering original region reuse optimisation is enabled). The overall memory required for the private storages sums up to 2.4 GiB.



(a) *Paraver* trace showing non-vectorized initialization and combination functions



(b) *Paraver* trace showing vectorized initialization and combination functions

6.6 Kernel initialization on write

Even though our implementation of reductions is mainly focused on dense array reductions, some applications have a sparse reduction pattern. When analysing how our implementation performed in such applications, we observed that most computational time was wasted on the allocation, initialisation and combination of the private storages. The reason behind is that, for sparse reductions, the required privatisation tends to be very large, while the accessed memory positions tend to be much fewer.

In an effort to extend our implementation to better support such scenarios, we realised that a complete initialisation of private storages was not an option anymore and that

a mechanism to control initialisation in a much finer way was required. We also were not looking forward to developing any complex mechanism to track the access regions within a private storage, as that would have a great penalisation on the critical path for the usual case we had chosen to prioritise: Dense array reductions. Instead, we were mostly interested in a way to provide a decent performance for such sparse applications so that our programming model was useful, but without affecting the base case we were most interested in.

We decided to implement this mechanism inside the Linux kernel, using the module mechanism to add it as an independent component. The idea behind this is to take advantage of the Kernel memory subsystem to initialise complete memory pages only when they are accessed for the first time and a *page fault* is generated. For the ones familiar with the *POSIX mmap*[20] system call, this is similar to using `MAP_ANONYMOUS` and `MAP_PRIVATE` (which initializes a copy-on-write[20] mapping with its contents initialized to zero), but with the particularity that the pre-initialized needs not be zero, but any other arbitrary value.

As far as the implementation is concerned, the developed kernel module is responsible for adding a special device file in the Linux filesystem. The user can open the newly added device, and then write the desired pre-initialization value and size (to be used for properly matching the *type*). At this point, the module proceeds to pre-initialize a whole memory page (usually 4 KiB of memory) with this value. Then, we can map this device to the desired memory region using a `mmap` *MAP_PRIVATE* mapping, effectively enabling the copy-on-write over the pre-initialized memory page.

After this set-up is complete, when the application tries to read the mapped region for the first time, the pre-initialized page will be read instead (with no extra cost). On the other hand, when the application tries to write that memory, the page fault service routines will allocate a new page and copy the pre-initialized page onto it. Then, the copy will be accessed for writing.

To sum up, we have developed a device that allows us to initialise memory pages to arbitrary values. When used in combination with the `mmap` system call, an efficient lazy initialisation mechanism can be implemented directly at kernel space. This mechanism is transparent to the *Nanos6* runtime, and hence it does not add any complexity to it. Moreover, the mechanism can easily be enabled or disabled by simply loading or unloading the module.

6.6.1 Proof of concept

For this proof of concept we were not able to use *MareNostrum4*, as loading a kernel module requires *root* privileges over the machine, which would be insecure to have on a machine of such characteristics. Instead, we had to move to a smaller development machine, whose characteristics are the following:

- Processors: 2x Intel(R) Xeon(R) CPU X5680 @ 3.33GHz
- # Cores: 12 cores (24 threads), 2 NUMA nodes
- Cache:
  - L1i: 32 KiB (4-way assoc.)
  - L1d: 32 KiB (8-way assoc.)
  - L2: 256 KiB (8-way assoc.)
  - L3: 12 MiB (16-way assoc.)
- Memory: 24 GiB (12 GiB per NUMA node)

As of the benchmark, we will be using the same benchmark we described in section 6.5.1, but this time we will make the reductions access a much wider memory region, in order to better exemplify a sparse problem. 12 reduction tasks will be instantiated, each performing 20 million accesses over a memory region of 200 MiB. The peculiarity will be that each of those memory regions will be offsetted so that its first half overlaps with the previous task's region, while its second half overlaps with the following task's region, as shown in figure 6.13. In addition, all tasks will be instantiated within a `weakreduction` task declared over the whole region. We use the `weakreduction` task to ensure that the private storages' size corresponds to the complete memory region.

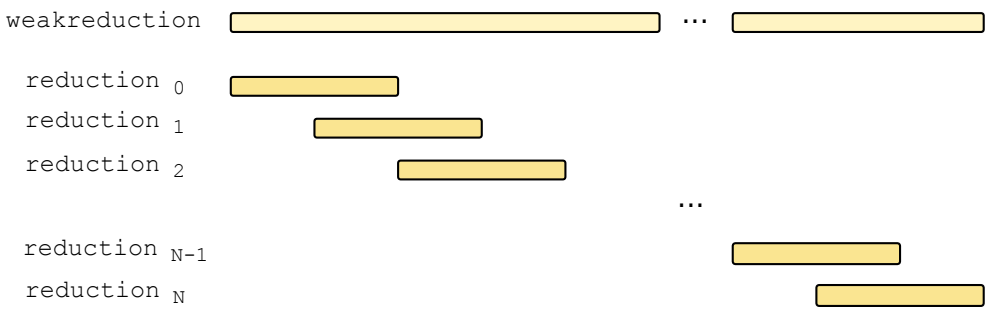
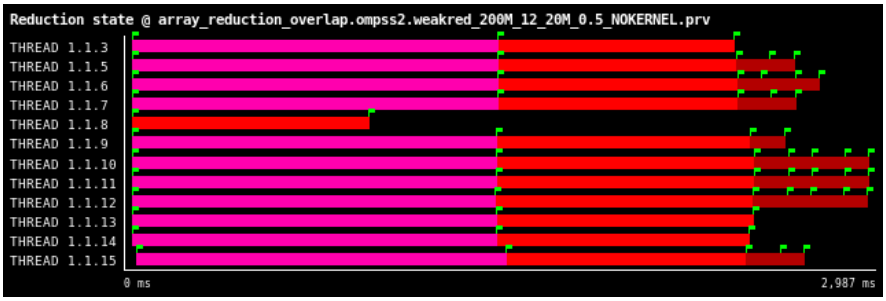


Figure 6.13: Benchmark `reduction` tasks registered memory regions

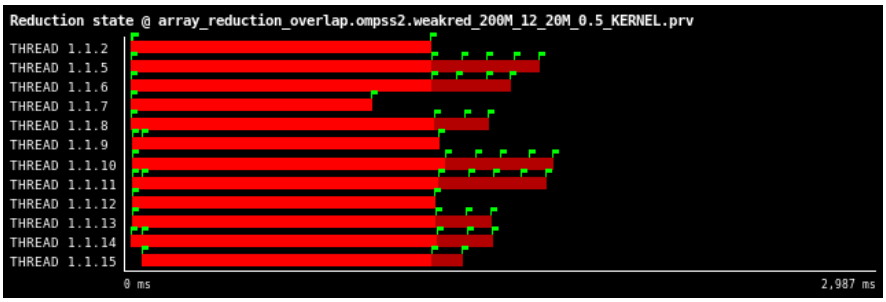
The overall memory required for the private storages would correspond to 12 times the whole memory region (over which the `weakreduction` is registered), totalling 15.6 GiB. Instead, as the kernel will only allocate and initialise the accessed memory, it is reduced to 2.4 GiB instead (the accessed 200 MiB for each CPU).

Figures 6.14a and 6.14b correspond to *Paraver* traces of a regular execution and an execution using the initialisation on the kernel. The bright red area corresponds to the execution of the task, while the dark red segments correspond to the combination of

the private storages and the pink correspond to the initialisation. In the kernel initialisation trace we do not observe the initialisation phase (in pink), as the initialisation is effectively performed when page faults are produced during the task execution, and thus the initialisation is comprehended by the bright red area (which is slightly larger than the baseline execution). We can see how having a fine-grained lazy initialisation can have a great impact on memory consumption and execution time.



(a) *Paraver* trace showing non-vectorized initialization and combination functions



(b) *Paraver* trace showing vectorized initialization and combination functions

7. Evaluation and results

This chapter covers the evaluation of the array reductions implementation presented in this work, as well as the optimisations that have been developed on top of it.

First, we introduce the benchmarking methodology, followed by platform and software environment in which the experiments where performed. Then, we provide an in-depth analysis of four benchmarks using array reductions. Finally, we discuss the obtained results.

7.1 Benchmark methodology

For each of the benchmarks presented in this chapter, our implementation is compared to a reference parallelisation of the benchmark, most times written in *OpenMP*.

In all analysed benchmarks where a reference *OpenMP* version is provided, we have found such version to be unoptimised. Therefore, we have been forced to improve the *OpenMP* parallelisations in order to present a fair comparison. Unfortunately, we have not been able to compare our implementation to *OpenMP* task reductions. Being a recently added feature, it is still not implemented in the reference vendor implementations.

As far as the benchmarking conditions are concerned, we have taken all considerations to provide valid and reproducible results.

First, we have ensured that all benchmarks were executed with exclusiveness of the node. That is, node resources were not shared with other processes or users during the performance measures. To achieve this, the Simple Linux Utility for Resource Management (SLURM) workload management system and its queuing system was used.

In addition, we have systematically guaranteed that at least five repetitions were run for each configuration in the benchmark. Then, the measured times for each of the five executions were averaged in order to minimise the effect of the measure variability between executions.

7.2 Environment

The evaluation of the presented implementation has been carried out on the *MareNostrum4* supercomputer found at the *BSC*.

Each compute node in *MareNostrum4* is equipped with 2 sockets of Intel Xeon Platinum 8160 CPUs, with 24 cores each, totalling 48 cores per node and 96 GB of main memory (2 GB per core).

As far as the memory hierarchy is concerned, each socket forms a NUMA node and all the cores that belong to the socket share the L3 cache. The other cache levels are private for each core. Both processors are connected by using a *QPI*[15] connection, forming a NUMA shared memory node.

Even though these processors support *HyperThreading*, it is disabled in this machine, and therefore there is only one thread per core.

We have tried to reflect the previous node topology when obtaining performance measurements. In this sense, all strong scalability plots shown in the following subsections will be following the same pattern: Starting from a sequential execution, the number of CPUs will be doubled at each step up to 16. Then, an execution filling the NUMA node will be provided (24 CPUs) and, finally, an addition execution using the whole node.

Details of this cluster node are shown in table 7.1.

#NUMA nodes	2
Memory/NUMA node	48GB
#Sockets	2
#Cores/socket	24
#Threads/core	1
#Total threads	48
L3 size	33MB
L2 size	1MB
L1d size	32KB
L1i size	32KB

Table 7.1: *MareNostrum4* node summary

Software	**Version**
GNU C/C++ compiler	7.2.0
Intel C/C++ compiler	18.0.5

Table 7.2: Used software versions

Table 7.2 shows the software and the versions used in our experiments. All applications have been compiled using the `-O2` optimisation level and enabling auto-vectorisation optimisations.

7.3 Matrix multiplication

Matrix multiplication is a classic yet important linear algebra benchmark for its relevance as a kernel in many scientific applications.

$C = A \times B$ matrix multiplication is an embarrassingly parallel benchmark, as each cell in the result matrix C is computed as the dot product between a row of the matrix A and a column of the matrix B, which is independent of the computation for any other cell, and thus each element can be computed completely in parallel.

However, being it a memory-bound problem, blocking or tiling techniques are often used to boost the performance by having a better reuse in the cache hierarchy, as illustrated in figure 7.1. When this is the case, the effective parallelism of the benchmark is given by the number of blocks that can be computed in parallel for the result matrix C:

$$\text{parallelism} = \frac{N}{B_y} \times \frac{M}{B_x} \tag{7.1}$$

In a blocked matrix multiplication we find an array reduction in the multiplication of all blocks of A and B matrices that correspond to the same block in the C matrix.

For this benchmark, we compare the performance of an *OmpSs-2* parallelisation using `inout` dependences between all blocks belonging in the same chain (i.e. contributing to the same result matrix block) versus another *OmpSs-2* parallelisation, this time using a `reduction` to specify the dependences between the blocks. Both codes can be found in the appendix B.1.

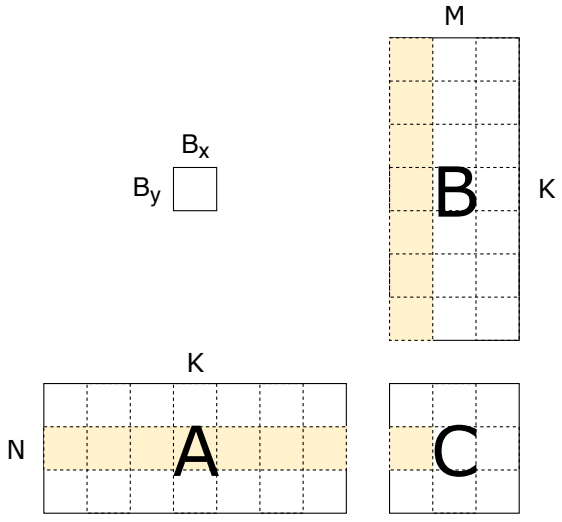


Figure 7.1: Matrix multiplication with blocking

The benchmark execution conditions are listed as follows:

- **Problem size:** 1024 **(N)** 1024 **(M)** 8192 **(K)**
- **Block size:** 256×256, 128×128, 64×64
- **Number of CPUs:** 48

Figure 7.2 shows the obtained results for this benchmark. At first glimpse, we can see how the `reduction` version outperforms the `inout` version for all block sizes. In order to explain the underlying reason we are going to classify the tested block sizes in 3 partitions:

- Too coarse-grained
- Suitable block size
- Too fine-grained

Too coarse-grained block sizes are those that do not allow enough parallelism (as defined by equation 7.1). In our experimentation, block-sizes *256* and *512* limit the parallelism to *16* and *4* respectively (for the `inout` version). On the other side, block sizes of *32* and *64* fall into the *too fine-grained* categorization. In those, there is plenty of parallelism, but the amount of tasks to create is so big (2^{18} and 2^{15} respectively), and the work to do by each task so small that the creator can not keep up instantiating tasks at the rate they are consumed, resulting in some threads having no work. This problem is known as having a *slow creator*. Finally, we would expect to find a *sweet spot* where the granularity is not too fine, but there is enough parallelism for all available resources. This point, however, depends on both the problem and the machine, and either does not exist or requires fine-tuning to be found.

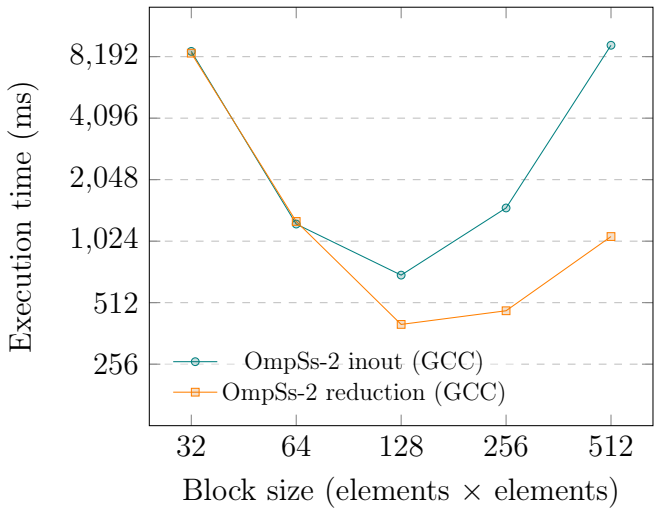


Figure 7.2: Matrix multiplication @ *MareNostrum4*

Reductions can help to make this *sweet-point* a wider region, making the fine-tuning less crucial while also making it more portable to other machines or problem sizes, and it does so just by relaxing the dependence. Relaxing a dependence (and thus providing more parallelism) will never penalize the performance, even if it is redundant. On the other hand, it will certainly help when the task creation rate and cache hierarchy require a bigger block-size.

When the granularity is too fine, both versions suffer evenly, but, on the other hand, the `reduction` version is much more robust to coarser block-sizes, as it is able to extract parallelism where the `inout` version can not. This robustness can be clearly seen in the previous plot by comparing the difference between choosing different block-sizes in both versions: The `inout` version is much more sensible on picking the right block-size than the `reduction` version is.

7.4 Two Point Angular Correlation Function

The TPACF is a metric used to statistically analyse the spatial distribution of observed astronomical bodies. The function describes the probability of finding two points separated by a given angular distance. This benchmark computes the angular correlation function for a data set of astronomical bodies, and does so by computing the distance between all pairs of input for then generating a histogram summary of the observed distances.[26,28]

The TPACF is part of the *Parboil* benchmark suite. The *Parboil* benchmarks are a set of throughput computing applications useful for studying the performance of throughput computing architecture and compilers. The benchmarks come from many different scientific fields including image processing, biomolecular simulation, fluid dynamics and astronomy. It would not be accurate to call any of the *Parboil* benchmarks a *complete application* in the typical sense, although several of the benchmarks are in fact run as standalone tools in some situations.[26,28]

Each benchmark in the *Parboil* suite includes several implementations. Some are provided as readable base implementations, while others are target specific architectures or parallel programming models such as *OpenMP OpenCL* or *CUDA*.[26,28]

In this benchmark we compare the performance of an *OmpSs-2* parallelisation of TPACF versus the reference *OpenMP* parallelisation in *Parboil*. Figure 7.3 shows the obtained results for a medium-sized problem composed of 100 random files with 4096 points each (*medium* dataset provided in the *Parboil* suite).

As we can see in the previous chart, the original *Parboil OpenMP* parallelisation performs much worse than the *OmpSs-2* parallelisation. However, we believe that this comparison is not fair, as there are better mechanisms available in *OpenMP* to provide

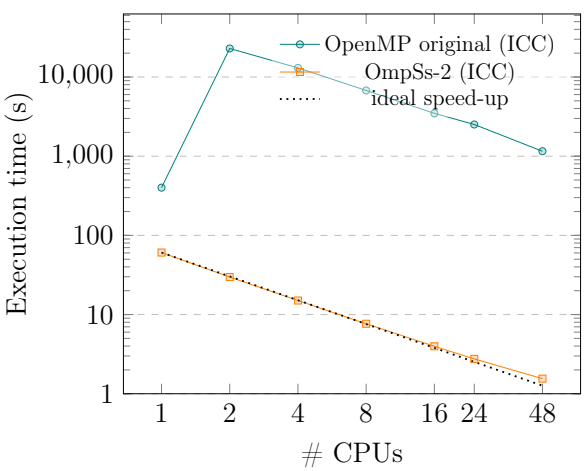


Figure 7.3: TPACF @ *MareNostrum4* (original)

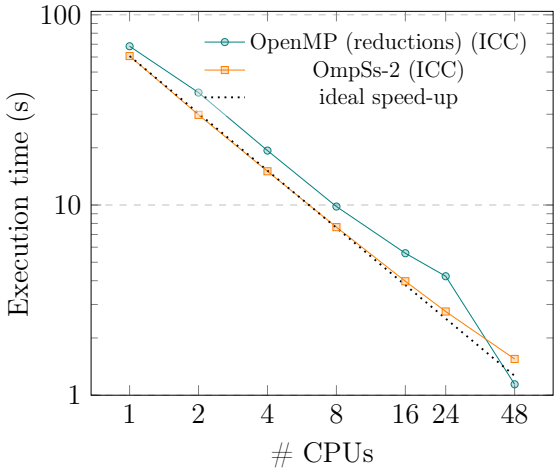


Figure 7.4: TPACF @ *MareNostrum4* (optimised)

a better parallelisation of this benchmark. For this reason, we have decided to improve the *OpenMP* parallelisation and provide a better comparison. Figure 7.4 show the results of the new parallelisation as compared to *OmpSs-2*. In appendix B.2 the different parallelisations used for this benchmark are provided.

From the performance difference between the two *OpenMP* versions, we can clearly perceive the benefits of providing support for array reductions in the underlying programming model.

When comparing the *OmpSs-2* version with the optimised *OpenMP* version, we can see that *OmpSs-2* still performs better than *OpenMP* in most scenarios, except when 48 CPUs are used. In that scenario, the *OpenMP* version is capable of obtaining a super-linear speed-up. We are unaware of which optimisation is performed by the *Intel* runtime to obtain this speed-up.

Besides, we can see how our *OmpSs-2* implementation has a good scalability both inside and outside the NUMA node.

7.5 K-means

K-means is a well known clustering algorithm extensively used in the field of data-mining and elsewhere.[8]

In k-means, a data object is comprised of several values, called features. By dividing a cluster of data objects into K sub-clusters, k-means represents all the data objects by the mean values or centroids of their respective sub-clusters. The initial cluster centre for each sub-cluster is randomly chosen or derived from some heuristic. In each iteration, the algorithm associates each data object with its nearest centre, based on some chosen distance metric. The new centroids are calculated by taking the mean of all the data objects within each sub-cluster respectively. The algorithm iterates until no data objects move from one sub-cluster to another.[8]

This k-means benchmark is part of the *Rodinia* benchmark suite. This suite is addressed to provide benchmarks for both general-purpose CPU architectures and heterogeneous systems that incorporate accelerators. For this line, the suite provides multiple versions of each benchmark optimised for different systems. In particular, an *OpenMP* parallelisation of each benchmark is provided.

In this benchmark we compare the performance of an *OmpSs-2* parallelisation of the k-means algorithm versus the reference version as provided by *Rodinia*. Similar to the previous benchmark, the *OpenMP* version provided by the *Rodinia* suite does not use *OpenMP* reductions. Instead, a user-driven reduction is performed. For an extended comparison, we have provided an additional parallelisation using *OpenMP* reductions. More details and the code for the different versions can be seen in appendix B.3.

Figure 7.5 shows the obtained results for a k-means clustering performed over 10^6 points in a 34-dimensional space using floating point coordinates.

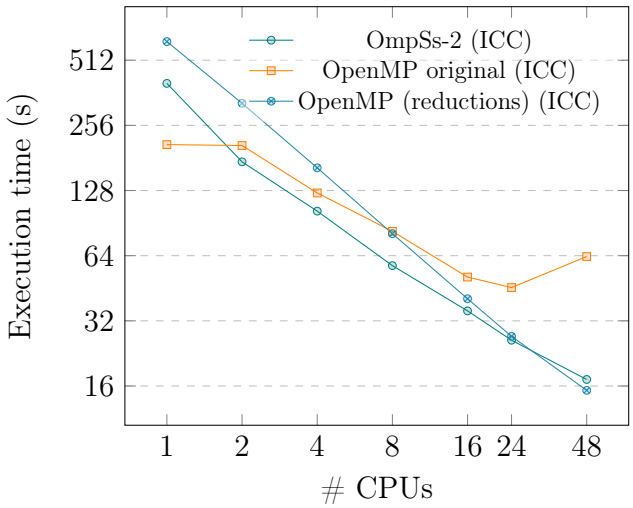


Figure 7.5: Kmeans @ *MareNostrum4*

When we compare the original *OpenMP* parallelisation and the new parallelisation using reductions, we can observe some differences in their behaviour. First, we can see how the original user-driven reduction found in the original version performs better when few CPUs are used, probably because the overheads of using *OpenMP* reductions do not compensate for the benefits they provide. When the number of CPUs is 8, the performance obtained in both parallelisations is equivalent, which indicates that the overheads and benefits are balanced out. Finally, as more CPUs are used, the parallelisation using *OpenMP* reductions outperforms the one provided in *Rodinia*, illustrating a situation where the benefits greatly exceed the overheads.

Besides, if we compare the parallelisation using *OpenMP* reductions with our implementation in *OmpSs-2*, we can see how our implementation is capable of obtaining a lower execution time in any configuration within the NUMA node, and a slightly higher one outside it.

7.6 Histogram

This benchmark consists in a straightforward histogramming operation that accumulates the number of occurrences of each output value in the input data set. The output histogram is a two-dimensional matrix of integer bins that saturate at 255[28]. This benchmark is part of the *Parboil* benchmark suite.

For this benchmark we have used the input sets provided in the *Parboil* suit, which correspond to a particular application setting in silicon wafer verification. In particular, the dimensions of the histogram are (256 W x 8192 H) and the input set follows a roughly Gaussian distribution, centred in the output histogram[28].

The benchmark simulates the computation of many histograms one after the other. Even though the input data is completely independent, the two-dimensional matrix where the histogram is reused between computations, and hence it must be cleared at the beginning of each histogram computation.

Unfortunately, due to the poor parallelisation provided for the reference *OpenMP* implementation, we have not been able to complete its executions within the time restrictions enforced in the *MareNostrum4* queuing system (>2 hours for a single execution step). For this reason, we do not provide performance measurements for the version supplied in *Parboil*, but rather provide the results for an improved *OpenMP* parallelisation we have developed using array reductions. Both *OpenMP* versions, as well as the *OmpSs-2* parallelisation, can be found in appendix B.4.

Figure 7.6 shows the strong scalability results obtained for *OpenMP* and *OmpSs-2* when resolving a medium-sized problem, in which 1000 histograms are computed (*medium* dataset provided in the *Parboil* suite).

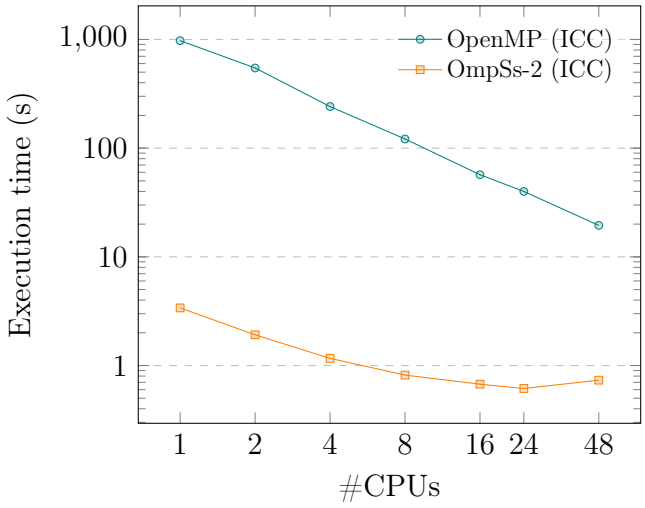


Figure 7.6: Histogram @ *MareNostrum4*

In this benchmark we observe a great difference between the two versions. After analysing the behaviour of the benchmark using *Paraver* traces, we have discovered that the *OmpSs-2* parallelisation is taking advantage of the *early beginning* optimisation presented in section 6.2 to compute the histograms in parallel. That is, while an iteration is being computed over the result two-dimensional matrix, the optimisation allows subsequent histograms to allocate the reduction private storages and start computing the histogram on them. Then, as soon as the computation is finished and the result matrix becomes available, the reduction is combined onto it.

This process is extended for all histograms to compute, providing enough parallelism for all CPUs during the whole execution. With it, the execution leads to a single dependence chain, formed by the combination of the histogram reductions interleaved with the *memset* operations used to clear the matrix. We can see this behaviour in the *Paraver* trace shown in figure 7.7. In it, we can see two differentiated phases: first, the parallel region where all histograms are being computed, then, a single dependence chain formed by the remaining reduction combinations.

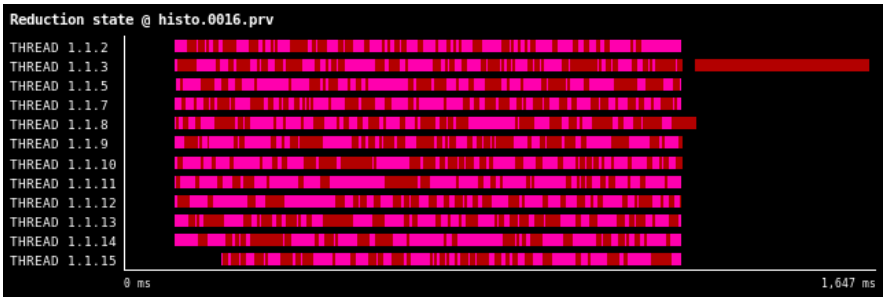


Figure 7.7: *Early beginning* on histogram

In this benchmark, the *early beginning* optimisation allows a better use of the available resources in the system, thus providing a competitive advantage over the *OpenMP* version. However, this optimisation forces the runtime to maintain a different privatisation set for every ongoing histogram computation. If the memory to privatise was larger and the histogram computations slower, this process could end up consuming all memory in the system.

8. Conclusions

In this project, we have accomplished to provide a flexible scheme for computing parallel reductions of arrays in the context of the *OmpSs-2* programming model.

In order to achieve this purpose, an extension of the *OmpSs-2* programming model has been proposed, enhancing the existent `reduction` clause to support flexible reductions of arrays. The features supported in this extension include overlaps between reductions and the combination of reductions at parts, known as *partial combination*. We have also introduced the `weakreduction` clause, necessary to support nested reductions efficiently and enable further run-time optimisations.

In order to assess the proposed extension, we have developed a compliant reference implementation using the *Mercurium* compiler and the *Nanos6* runtime. With respect to its design, several strategies have been considered, discussing their differences and the applications each best suit. Then, the complete design of the implementation is presented.

This implementation has been used as a baseline from which we have explored a wide range of optimisations of different complexities and applicability. This report summarises the benefits and weaknesses of each optimisation, while explaining the background idea that has inspired them. We believe this information will be very useful to developers of task-based programming models as well as to the community.

Such feature would not be of much utility if the provided implementation were not competitive performance-wise. After all, the final goal of the *OmpSs-2* programming model is to facilitate programming for parallel architectures without compromising the performance. For this reason, the performance of our implementation has been evaluated using four distinct benchmarks. The results of this evaluation have showed how the provided implementation outperforms state-of-the-art *OpenMP* parallelisations in most scenarios.

To conclude, the work accomplished in this project fulfills its initial goals and beyond, providing a well-performing mechanism to simplify the computation of array reductions, and evaluating it on a range of relevant applications.

9. Future work

Even though the project has fulfilled its initial goals and the obtained results are satisfactory, there are still many aspects related to the optimisation of the reduction pattern left to be dealt with.

Following our line of work, the first extension we propose consists in improving the provided implementation by parallelising the combination process. This would help providing a better support to reductions of big arrays.

As we have already discussed in previous sections, controlling the amount of resources dedicated to the reduction computation is a valuable upgrade. A contribution on this topic would be worthwhile and innovative.

Related to the previous point, the algorithm that assigns storages to CPUs in the *dynamic privatisation* strategy could be refined. In this sense, reassigning storages to CPUs that have already used them would lead to a greater memory affinity.

Something as useful would be to generalise the privatisation mechanism in reductions so that it could be used in other contexts. For instance, WaR dependences could be removed by privatising the input data, allowing a successor writer to begin immediately.

As soon as task reductions are implemented by the major *OpenMP* vendors, we would like to extend this project's evaluation chapter so that it includes them. With them, a much more accurate comparison can be provided.

Finally, we believe that an interesting contribution to the *OmpSs-2* programming model would be to support the execution of task reductions on heterogeneous systems. Accelerators like GPUs are currently getting a lot of attention in the field of HPC. For this, programming models like *OpenMP* or *OmpSs-2* are putting efforts in being able to facilitate the execution on those devices. We believe that executing reduction tasks in the accelerators could provide enormous performance benefits.

10. References

[1] G.D. Kim (originator). *Encyclopedia of Mathematics. Seidel method*. Feb. 2011. URL: http://www.encyclopediaofmath.org/index.php?title=Seidel_method&oldid=18222 (visited on 12/18/2018) (cit. on p. 42).

[2] OpenMP Architecture Review Board. *OpenMP Application Programming Interface: Version 5.0*. Nov. 2018. URL: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf (visited on 11/25/2018) (cit. on p. 21).

[3] OpenMP Architecture Review Board. *OpenMP FAQ*. June 2018. URL: https://www.openmp.org/about/openmp-faq/#WhatIs (visited on 11/26/2018) (cit. on p. 1).

[4] BSC-CNS. *Mercurium*. URL: https://pm.bsc.es/mcxx (visited on 12/19/2018) (cit. on p. 14).

[5] BSC-CNS. *OmpSs-2*. URL: https://pm.bsc.es/ompss-2 (visited on 12/18/2018) (cit. on pp. 1, 5–14, 31).

[6] BSC-CNS. *The OmpSs Programming Model*. URL: https://pm.bsc.es/ompss (visited on 12/19/2018) (cit. on pp. 1, 5).

[7] Barcelona Supercomputing Center. *MareNostrum Technical Information*. Dec. 2018. URL: https://www.bsc.es/marenostrum/marenostrum/technical-information (visited on 12/18/2018) (cit. on p. 52).

[8] S. Che et al. "Rodinia: A benchmark suite for heterogeneous computing". In: *2009 IEEE International Symposium on Workload Characterization (IISWC)*. Oct. 2009, pp. 44–54. DOI: 10.1109/IISWC.2009.5306797 (cit. on p. 62).

[9] Jan Ciesko et al. *Towards Task-Parallel Reductions in OpenMP*. Springer International Publishing, 2015 (cit. on p. 3).

[10] CppReference. *Reference declaration*. Dec. 2015. URL: http://en.cppreference.com/w/cpp/language/reference (visited on 12/27/2018) (cit. on pp. 9, 12).

[11] R. H. Dennard et al. "Design of ion-implanted MOSFET's with very small physical dimensions". In: *IEEE Journal of Solid-State Circuits* 9.5 (Oct. 1974), pp. 256–268. ISSN: 0018-9200. DOI: 10.1109/JSSC.1974.1050511 (cit. on p. 1).

[12] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard, Version 3.1*. June 2015. URL: https://www.mpi-forum.org/ (visited on 11/26/2018) (cit. on p. 1).

[13] Philip Ginsbach and Michael F. P. O'Boyle. "Discovery and Exploitation of General Reductions: A Constraint Based Approach". In: *Proceedings of the 2017 International Symposium on Code Generation and Optimization*. CGO '17. Austin, USA: IEEE Press, 2017, pp. 269–280. ISBN: 978-1-5090-4931-8 (cit. on p. 23).

[14] BSC Tools group. *Extrae*. URL: `https://tools.bsc.es/extrae` (visited on 12/28/2018) (cit. on p. 15).

[15] Intel. *An Introduction to the Intel QuickPath Interconnect*. Jan. 2009. URL: `http://www.intel.com/content/dam/doc/white-paper/quick-path-interconnect-introduction-paper.pdf` (visited on 01/07/2018) (cit. on p. 57).

[16] Intel. *CilkPlus*. URL: `https://www.cilkplus.org/` (visited on 12/15/2018).

[17] Seonggun Kim, Hwansoo Han, and Kwang-Moo Choe. "Region-based parallelization of irregular reductions on explicitly managed memory hierarchies". In: *The Journal of Supercomputing* 56.1 (Apr. 2011), pp. 25–55. ISSN: 1573-0484. DOI: `10.1007/s11227-009-0340-3` (cit. on p. 23).

[18] Timothy. G. Mattson, Beverly A. Sanders, and Berna L. Massingill. *A Pattern Language for Parallel Programming*. Addison Wesley Software Patterns Series, 2004. (Visited on 11/29/2018) (cit. on p. 3).

[19] Timothy. G. Mattson, Beverly A. Sanders, and Berna L. Massingill. *Reduction Design Pattern*. Sept. 2005. URL: `http://www.cise.ufl.edu/research/ParallelPatterns/PatternLanguage/SupportingStructures/Reduction.htm` (visited on 11/29/2018) (cit. on p. 3).

[20] *mmap(2) Linux Programmer's Manual*. 4.16. Apr. 2018. URL: `http://man7.org/linux/man-pages/man2/mmap.2.html` (visited on 12/18/2018) (cit. on p. 53).

[21] BSC Programming Models. *OmpSs-2 User Guide*. URL: `https://pm.bsc.es/ftp/ompss-2/doc/user-guide/nanos6/common.html` (visited on 12/28/2018) (cit. on p. 15).

[22] G. E. Moore. "Cramming More Components Onto Integrated Circuits". In: *Proceedings of the IEEE* 86.1 (Jan. 1998), pp. 82–85. ISSN: 0018-9219. DOI: `10.1109/JPROC.1998.658762` (cit. on p. 1).

[23] Ferran Pallarès. "Extending OmpSs programming model with task reductions: A compiler and runtime approach". Bachelor's Thesis. Facultat d'Informàtica de Barcelona, Jan. 2017. URL: `http://hdl.handle.net/2117/101603` (visited on 11/20/2018) (cit. on pp. 3, 10, 18, 24, 25).

[24] *Programming Languages - C++*. Section 3.10, page 74. ISO/IEC, May 2013. URL: `https://isocpp.org/files/papers/N3690.pdf` (visited on 12/27/2018) (cit. on pp. 9, 12).

[25] C. Reddy, M. Kruse, and A. Cohen. "Reduction drawing: Language constructs and polyhedral compilation for reductions on GPUs". In: *2016 International Conference on Parallel Architecture and Compilation Techniques (PACT)*. Sept. 2016, pp. 87–97. DOI: `10.1145/2967938.2967950` (cit. on p. 24).

[26] Andy Schuh. *Parboil Benchmarks*. URL: http://impact.crhc.illinois.edu/Parboil/parboil.aspx (visited on 01/02/2018) (cit. on p. 60).

[27] ML Scott and WJ Bolosky. "False sharing and its effect on shared memory performance". In: *Proceedings of the USENIX Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS)*. Vol. 57. 1993. URL: https://www.usenix.org/legacy/publications/library/proceedings/sedms4/full_papers/bolosky.txt (cit. on p. 41).

[28] John A. Stratton et al. *Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing*. Tech. rep. IMPACT, University of Illinois at Urbana-Champaign, Mar. 2012. URL: http://impact.crhc.illinois.edu/Shared/Docs/impact-12-01.parboil.pdf (visited on 01/02/2018) (cit. on pp. 60, 63).

[29] Oreste Villa et al. *Effects of Floating-Point non-Associativity on Numerical Computations on Massively Multithreaded Systems*. Proceedings of Cray User Group Meeting (CUG), May 2009 (cit. on p. 3).

[30] Wikipedia. *Letter Frequency*. Dec. 2018. URL: https://en.wikipedia.org/wiki/Letter_frequency (visited on 12/20/2018) (cit. on p. 4).

A. Supported reduction operators

This section describes the supported reduction operators and their identifier symbol. Table A.1 shows the supported reduction identifiers and their initializer and combiner functions.

Identifier	Initializer	Combiner
`+`	`oss_out = 0`	`oss_out += oss_in`
`*`	`oss_out = 1`	`oss_out *= oss_in`
`-`	`oss_out = 0`	`oss_out += oss_in`
`&`	`oss_out = ~0`	`oss_out &= oss_in`
`|`	`oss_out = 0`	`oss_out |= oss_in`
`^`	`oss_out = 0`	`oss_out ^= oss_in`
`&&`	`oss_out = 1`	`oss_out = oss_in && oss_out`
`||`	`oss_out = 0`	`oss_out = oss_in || oss_out`
`max`	`oss_out` = least representable number in the type	`oss_out = oss_in > oss_out ? oss_in : oss_out`
`min`	`oss_out` = largest representable number in the type	`oss_out = oss_in < oss_out ? oss_in : oss_out`

Table A.1: Supported reduction operators

B. Benchmarks code

In this appendix we display the source code of each of the benchmarks discussed and analysed in chapter 7, including the different parallelisation strategies used.

These are the code listings that can be found in the following pages:

- Matrix multiplication
  - *OmpSs-2* parallelisation using `inout` dependences (B.1)
  - *OmpSs-2* parallelisation using reductions (B.2)
- TPACF
  - *OmpSs-2* parallelisation (B.3, B.4)
  - Original *OpenMP* parallelisation (B.5)
  - Optimised *OpenMP* parallelisation (B.6, B.7)
- K-means
  - *OmpSs-2* parallelisation (B.8)
  - Original *OpenMP* parallelisation (B.9)
  - Optimised *OpenMP* parallelisation (B.10)
- Histogram
  - *OmpSs-2* parallelisation (B.11)
  - Original *OpenMP* parallelisation (B.12)
  - Optimised *OpenMP* parallelisation (B.13)

B.1 Matrix multiplication

```
void matmul(const long N, const long TS,
           double (* restrict A)[N/TS][TS][TS],
           double (* restrict B)[N/TS][TS][TS],
           double (* restrict C)[N/TS][TS][TS])
{
    long const NB = N / TS;

    for (long k=0; k < NB; k++) {
        double *C_aux = (double*)C;
        #pragma oss task weakinout([NB*NB*TS*TS]C_aux) label(K iteration)
        for (long i=0; i < NB; i++) {
            for (long j=0; j < NB; j++) {
                double (* restrict A_block)[TS] = A[i][k];
                double (* restrict B_block)[TS] = B[k][j];
                double (* restrict C_block)[TS] =
                    ((double (*)[N/TS][TS][TS])C_aux)[i][j];
                double *C_block_aux = (double*)C_block;

                #pragma oss task inout([TS*TS]C_block_aux) label(matmul block)
                for (long ii=0; ii < TS; ii++) {
                    for (long jj=0; jj < TS; jj++) {
                        for (long kk=0; kk < TS; kk++) {
                            ((double (*)[TS])C_block_aux)[ii][jj]
                                += A_block[ii][kk] * B_block[kk][jj];
                        }
                    }
                }
            }
        }
    }
}
```

Listing B.1: *OmpSs-2* parallelisation of matrix multiplication (`inout` version)

```
void matmul(const long N, const long TS,
            double (* restrict A)[N/TS][TS][TS],
            double (* restrict B)[N/TS][TS][TS],
            double (* restrict C)[N/TS][TS][TS])
{
    long const NB = N / TS;

    for (long k=0; k < NB; k++) {
        double *C_aux = (double*)C;
        #pragma oss task weakreduction(+: [NB*NB*TS*TS]C_aux) label(K iteration)
        for (long i=0; i < NB; i++) {
            for (long j=0; j < NB; j++) {
                double (* restrict A_block)[TS] = A[i][k];
                double (* restrict B_block)[TS] = B[k][j];
                double (* restrict C_block)[TS] =
                    ((double (*)[N/TS][TS][TS])C_aux)[i][j];
                double *C_block_aux = (double*)C_block;

                #pragma oss task reduction(+: [TS*TS]C_block_aux) label(matmul block)
                for (long ii=0; ii < TS; ii++) {
                    for (long jj=0; jj < TS; jj++) {
                        for (long kk=0; kk < TS; kk++) {
                            ((double (*)[TS])C_block_aux)[ii][jj]
                                += A_block[ii][kk] * B_block[kk][jj];
                        }
                    }
                }
            }
        }
    }
}
```

Listing B.2: *OmpSs-2* parallelisation of matrix multiplication (`reduction` version)

B.2 TPACF

```
int main(int argc, char **argv)
{
    [...]

    // compute DD
    doCompute(data, npd, NULL, 0, 1, DD, nbins, binb);

    // loop through random data files
    for (rf = 0; rf < args.random_count; rf++)
    {
        // read random file
        #pragma oss task \
        label(io) \
        firstprivate(rf) \
        out(npr) \
        out(random[0; args.npoints])
        {
            pb_SwitchToTimer(&timers, pb_TimerID_IO);
            npr = readdatafile(params->inpFiles[rf+1], random, args.npoints);
            pb_SwitchToTimer(&timers, pb_TimerID_COMPUTE);
        }

        // compute RR
        #pragma oss task \
        label(compute RR) \
        in(npr, nbins) \
        in(random[0; args.npoints], binb[0; nbins + 1])
        weakreduction(+: [nbins + 1]RRS) \
        doCompute(random, npr, NULL, 0, 1, RRS, nbins, binb);

        // compute DR
        #pragma oss task \
        label(compute DR) \
        in(npd, npr, nbins) \
        in(data[0; args.npoints], random[0; args.npoints], binb[0; nbins + 1])
        weakreduction(+: [nbins + 1]DRS) \
        doCompute(data, npd, random, npr, 0, DRS, nbins, binb);
    }

    [...]
}
```

Listing B.3: *OmpSs-2* parallelisation of TPACF (main)

```
int doCompute(struct cartesian *data1, int n1, struct cartesian *data2,
          int n2, int doSelf, long long *data_bins,
          int nbins, float *binb)
{
    int i, j, k;
    if (doSelf) {
        n2 = n1;
        data2 = data1;
    }

    #pragma oss loop \
    label(doCompute) \
    chunksize(4) \
    private(i, j) \
    firstprivate(doSelf, n1, n2, nbins) \
    in(data1[0; (doSelf?n1-1:n1)], data2[(doSelf?1:0):  n2-1], binb[0; nbins]) \
    reduction(+: [nbins + 1]data_bins)
    for (i = 0; i < ((doSelf) ? n1-1 : n1); i++) {
        const register float xi = data1[i].x;
        const register float yi = data1[i].y;
        const register float zi = data1[i].z;

        for (j = ((doSelf) ? i+1 : 0); j < n2; j++) {
            [...]
        }
    }

    return 0;
}
```

Listing B.4: *OmpSs-2* parallelisation of TPACF (doCompute)

```
int doCompute(struct cartesian *data1, int n1, struct cartesian *data2,
          int n2, int doSelf, long long *data_bins,
          int nbins, float *binb)
{
    int i, j, k;
    if (doSelf) {
        n2 = n1;
        data2 = data1;
    }

    for (i = 0; i < ((doSelf) ? n1-1 : n1); i++) {
        const register float xi = data1[i].x;
        const register float yi = data1[i].y;
        const register float zi = data1[i].z;

        #pragma omp parallel for
        for (j = ((doSelf) ? i+1 : 0); j < n2; j++) {
            register float dot = xi * data2[j].x + yi * data2[j].y +
                zi * data2[j].z;

            // run binary search
            register int min = 0;
            register int max = nbins;
            register int k, indx;

            while (max > min+1) {
                k = (min + max) / 2;
                if (dot >= binb[k])
                    max = k;
                else
                    min = k;
            };

            #pragma omp critical
            if (dot >= binb[min])
                data_bins[min] += 1; /*k = min;*/
            else if (dot < binb[max])
                data_bins[max+1] += 1; /*k = max+1;*/
            else
                data_bins[max] += 1; /*k = max;*/
        }
    }

    return 0;
}
```

Listing B.5: Original *OpenMP* parallelisation of TPACF as found in *Parboil* suite

```
int main(int argc, char **argv)
{
[...]

    #pragma omp parallel private(rf) \
    reduction(+: DD[0:nbins + 1], RRS[0:nbins + 1], DRS[0:nbins + 1])
    {
        // compute DD
        doCompute(data, npd, NULL, 0, 1, DD, nbins, binb);

        // loop through random data files
        for (rf = 0; rf < args.random_count; rf++) {
            #pragma omp single
            {
                // read random file
                pb_SwitchToTimer(&timers, pb_TimerID_IO);
                npr = readdatafile(params->inpFiles[rf+1], random, args.npoints);
                pb_SwitchToTimer(&timers, pb_TimerID_COMPUTE);
            }

            // compute RR
            doCompute(random, npr, NULL, 0, 1, RRS, nbins, binb);

            // compute DR
            doCompute(data, npd, random, npr, 0, DRS, nbins, binb);

            #pragma omp barrier
        }
    }

[...]
}
```

Listing B.6: Optimised *OpenMP* parallelisation of TPACF using array reductions (main)

For the optimized *OpenMP* parallelisation, we have removed the `critical` *OpenMP* construct from the `doCompute` function, replaced the `parallel for` construct for a simple `for` construct and finally add a `nowait` clause in the `for` construct in order to eliminate its implicit barrier. The `parallel` construct has been moved outside the program main loop, as we can see in the listing B.6, which has required to add a `single` region and a synchronization `barrier`.

```
int doCompute(struct cartesian *data1, int n1, struct cartesian *data2,
          int n2, int doSelf, long long *data_bins,
          int nbins, float *binb)
{
    int i, j, k;
    if (doSelf) {
        n2 = n1;
        data2 = data1;
    }

    #pragma omp for nowait \
    for (i = 0; i < ((doSelf) ? n1-1 : n1); i++) {
        const register float xi = data1[i].x;
        const register float yi = data1[i].y;
        const register float zi = data1[i].z;

        for (j = ((doSelf) ? i+1 : 0); j < n2; j++) {
            [...]
        }
    }

    return 0;
}
```

Listing B.7: Optimised *OpenMP* parallelisation of TPACF using array reductions (doCompute)

B.3 Kmeans

```
float** kmeans_clustering(float **feature, /* in: [npoints][nfeatures] */
                          int nfeatures,
                          int npoints,
                          int nclusters,
                          float threshold,
                          int *membership) /* out: [npoints] */
{
    [...]
    int i, j, k, n=0, index, loop=0;
    int *new_centers_len; /* [nclusters]: no. of points in each cluster */
    float **new_centers; /* [nclusters][nfeatures] */
    float **clusters; /* out: [nclusters][nfeatures] */
    float old_delta, delta;
    [...]

    do {
        #pragma oss task inout(delta) out(old_delta) label(update) priority(100
        {
            old_delta = delta;
            delta = 0.0;
        }

        #pragma oss loop \
        label(loop) \
        private(i, j, index) \
        firstprivate(nclusters, nfeatures, npoints) \
        firstprivate(feature, membership) \
        in([nfeatures](clusters[idx]), idx = 0;nclusters) \
        reduction(+: delta) \
        reduction(+: [nclusters]new_centers_len) \
        reduction(+: [nclusters*nfeatures]new_centers_ptr)
        for (i=0; i<npoints; i++) {
            /* find the index of nestest cluster centers */
            index = find_nearest_point(feature[i],
                    nfeatures,
                    clusters,
                    nclusters);
            /* if membership changes, increase delta by 1 */
            if (membership[i] != index) {
                delta += 1.0;
            }
```

```
            /* assign the membership to object i */
            membership[i] = index;

            /* update new cluster centers : sum of all objects located
               within */
            new_centers_len[index]++;

            for (j=0; j<nfeatures; j++) {
                new_centers_ptr[index*nfeatures + j] += feature[i][j];
            }
        }

        #pragma oss task \
        label(recompute_centers) \
        in([nclusters]new_centers_len) \
        out([nfeatures](clusters[idx]), idx = 0;nclusters) \
        inout([nfeatures](new_centers[idx]), idx = 0;nclusters)
        /* replace old cluster centers with new_centers */
        for (i=0; i<nclusters; i++) {
            for (j=0; j<nfeatures; j++) {
                if (new_centers_len[i] > 0)
                    new_centers[i][j] = new_centers[i][j] / new_centers_len[i];
                clusters[i][j] = 0.0;
            }
            new_centers_len[i] = 0.0;
        }

        float **aux = clusters;
        clusters = new_centers;
        new_centers = aux;
        new_centers_ptr = new_centers[0];

        #pragma oss taskwait in(old_delta) priority(100) label(wait_delta)

    } while (delta > threshold && loop++ < 500);
    #pragma oss taskwait

    [...]
}
```

Listing B.8: *OmpSs-2* parallelisation of kmeans

```
float** kmeans_clustering(float **feature, /* in: [npoints][nfeatures] */
                          int nfeatures,
                          int npoints,
                          int nclusters,
                          float threshold,
                          int *membership) /* out: [npoints] */
{
    [...]
    int i, j, k, n=0, index, loop=0;
    int *new_centers_len; /* [nclusters]: no. of points in each cluster */
    float **new_centers; /* [nclusters][nfeatures] */
    float **clusters; /* out: [nclusters][nfeatures] */
    float delta;

    int nthreads = omp_get_num_threads();
    int **partial_new_centers_len;
    float ***partial_new_centers;
    [...]

    do {
        delta = 0.0;
        #pragma omp parallel \
        shared(feature, clusters, membership, partial_new_centers, partial_new_centers_len)
        {
            int tid = omp_get_thread_num();
            #pragma omp for \
            private(i, j, index) \
            firstprivate(npoints, nclusters, nfeatures) \
            schedule(static) \
            reduction(+: delta)
            for (i=0; i<npoints; i++) {
                /* find the index of nestest cluster centers */
                index = find_nearest_point(feature[i],
                        nfeatures,
                        clusters,
                        nclusters);
                /* if membership changes, increase delta by 1 */
                if (membership[i] != index) delta += 1.0;

                /* assign the membership to object i */
                membership[i] = index;

                /* update new cluster centers : sum of all objects located
                   within */
```

```
                partial_new_centers_len[tid][index]++;
                for (j=0; j<nfeatures; j++)
                    partial_new_centers[tid][index][j] += feature[i][j];
            }
        } /* end of #pragma omp parallel */

        /* let the main thread perform the array reduction */
        for (i=0; i<nclusters; i++) {
            for (j=0; j<nthreads; j++) {
                new_centers_len[i] += partial_new_centers_len[j][i];
                partial_new_centers_len[j][i] = 0.0;
                for (k=0; k<nfeatures; k++) {
                    new_centers[i][k] += partial_new_centers[j][i][k];
                    partial_new_centers[j][i][k] = 0.0;
                }
            }
        }

        /* replace old cluster centers with new_centers */
        for (i=0; i<nclusters; i++) {
            for (j=0; j<nfeatures; j++) {
                if (new_centers_len[i] > 0)
                    clusters[i][j] = new_centers[i][j] / new_centers_len[i];
                new_centers[i][j] = 0.0; /* set back to 0 */
            }
            new_centers_len[i] = 0; /* set back to 0 */
        }

    } while (delta > threshold && loop++ < 500);

    [...]
}
```

Listing B.9: *OpenMP* original parallelisation of kmeans as found in *Rodinia* suite

```
float** kmeans_clustering(float **feature, /* in: [npoints][nfeatures] */
                          int nfeatures,
                          int npoints,
                          int nclusters,
                          float threshold,
                          int *membership) /* out: [npoints] */
{
    [...]
    int i, j, k, n=0, index, loop=0;
    int *new_centers_len; /* [nclusters]: no. of points in each cluster */
    float **new_centers; /* [nclusters][nfeatures] */
    float **clusters; /* out: [nclusters][nfeatures] */
    float delta, old_delta;
    [...]

    #pragma omp parallel \
    shared(feature, clusters, membership, new_centers, new_centers_ptr, new_centers_len) \
    shared(old_delta, delta) \
    firstprivate(loop)
    do {
        #pragma omp for \
        schedule(guided, npoints/(num_omp_threads*8)) \
        private(i, j, index) \
        firstprivate(nclusters, nfeatures, npoints, feature, membership) \
        reduction(+: new_centers_len[0:nclusters]) \
        reduction(+: new_centers_ptr[0:nclusters*nfeatures]) \
        reduction(+: delta)
        for (i=0; i<npoints; i++) {
            /* find the index of nestest cluster centers */
            index = find_nearest_point(feature[i],
                   nfeatures,
                   clusters,
                   nclusters);
            /* if membership changes, increase delta by 1 */
            if (membership[i] != index) {
                delta += 1.0;
            }

            /* assign the membership to object i */
            membership[i] = index;

            /* update new cluster centers : sum of all objects located
               within */
            new_centers_len[index]++;
```

```
            for (j=0; j<nfeatures; j++) {
                new_centers_ptr[index*nfeatures + j] += feature[i][j];
            }
        }

        #pragma omp single
        {
            /* replace old cluster centers with new_centers */
            for (i=0; i<nclusters; i++) {
                for (j=0; j<nfeatures; j++) {
                    if (new_centers_len[i] > 0)
                        new_centers[i][j] = new_centers[i][j] / new_centers_len[i];
                    clusters[i][j] = 0.0;
                }
                new_centers_len[i] = 0.0;

                old_delta = delta;
            }

            float **aux = clusters;
            clusters = new_centers;
            new_centers = aux;
            new_centers_ptr = new_centers[0];
        }

    } while (old_delta > threshold && loop++ < 500);

    [...]
}
```

Listing B.10: *OpenMP* optimised parallelisation of kmeans benchmark using array reductions

B.4 Histogram

```
void satadd_reducer(unsigned char *oss_out, unsigned char *oss_in) {
    uint16_t tmp = ((uint16_t) *oss_out) + *oss_in;
    *oss_out = (tmp < UINT8_MAX) ? tmp : UINT8_MAX;
}

int main(int argc, char* argv[]) {
    [...]

    unsigned int img_width, img_height;
    unsigned int histo_width, histo_height;

    unsigned int* img =
        (unsigned int*)malloc(img_width*img_height*sizeof(unsigned int));
    unsigned char* histo =
        (unsigned char*)calloc(histo_width*histo_height, sizeof(unsigned char));

    [...]

    #pragma oss declare \
        reduction(satadd : unsigned char : satadd_reducer(&oss_out, &oss_in)) \
        initializer(oss_priv = 0U)

    int iter;
    for (iter = 0; iter < numIterations; iter++){
        #pragma oss task inout([histo_width*histo_height]histo) \
        memset(histo, 0, histo_height*histo_width*sizeof(unsigned char));

        unsigned int i;
        #pragma oss loop \
            private(i) firstprivate(img_width, img_height) \
            shared(img) \
            reduction(sat_add: [histo_width*histo_height]histo) \
        for (i = 0; i < img_width*img_height; ++i) {
            const unsigned int value = img[i];

            if (histo[value] < UINT8_MAX) {
                ++histo[value];
            }
        }
    }
    #pragma oss taskwait
    [...]
}
```

Listing B.11: *OmpSs-2* parallelisation of histogram

```
int main(int argc, char* argv[]) {
    [...]

    unsigned int img_width, img_height;
    unsigned int histo_width, histo_height;

    unsigned int* img =
        (unsigned int*)malloc(img_width*img_height*sizeof(unsigned int));
    unsigned char* histo =
        (unsigned char*)calloc(histo_width*histo_height, sizeof(unsigned char));

    [...]

    int iter;
    for (iter = 0; iter < numIterations; iter++){
        memset(histo, 0, histo_height*histo_width*sizeof(unsigned char));

        unsigned int i;
        #pragma omp parallel for
        for (i = 0; i < img_width*img_height; ++i) {
            const unsigned int value = img[i];

            #pragma omp critical
            if (histo[value] < UINT8_MAX) {
                ++histo[value];
            }
        }
    }

    [...]
}
```

Listing B.12: *OpenMP* original parallelisation of histogram as found in *Parboil* suite

```
void satadd_reducer(unsigned char *omp_out, unsigned char *omp_in) {
    uint16_t tmp = ((uint16_t) *omp_out) + *omp_in;
    *omp_out = (tmp < UINT8_MAX) ? tmp : UINT8_MAX;
}

int main(int argc, char* argv[]) {
    [...]

    unsigned int img_width, img_height;
    unsigned int histo_width, histo_height;

    unsigned int* img =
        (unsigned int*)malloc(img_width*img_height*sizeof(unsigned int));
    unsigned char* histo =
        (unsigned char*)calloc(histo_width*histo_height, sizeof(unsigned char));

    [...]

    #pragma omp declare \
        reduction(satadd : unsigned char : satadd_reducer(&omp_out, &omp_in)) \
        initializer(omp_priv = 0U)

    #pragma omp parallel
    {
        int iter;
        for (iter = 0; iter < numIterations; iter++){
            #pragma omp single
            memset(histo, 0, histo_height*histo_width*sizeof(unsigned char));

            unsigned int i;
            #pragma omp for reduction(satadd: histo[0:histo_width*histo_height])
            for (i = 0; i < img_width*img_height; ++i) {
                const unsigned int value = img[i];

                if (histo[value] < UINT8_MAX) {
                    ++histo[value];
                }
            }
        }
    }
    [...]
}
```

Listing B.13: *OpenMP* optimised parallelisation of histogram benchmark using array reductions